Komponenttipohjaisen micro frontend -arkkitehtuurin suunnittelu: tapaus Visma Tampuuri Oy:n Asukassivut

Jouni Männistö

Helsinki 03.06.2021

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen osasto

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta/Osasto – Fakultet/Sektion – Faculty/Section	Koulutusohjelma – Studieprogram – Study Program	
Matemaattis-luonnontieteellinen tiedekunta/ Tietojenkäsittelytieteen osasto	Tietojenkäsittelytieteen maisteriohjelma	
Tekijä – Författare – Author		
Jouni Männistö		
Työn nimi – Arbetets titel – Title		
Komponenttipohjaisen micro frontend -arkkitehtuurin suunnittelu: tapaus Visma Tampuuri Oy:n Asukassivut		
Ohjaajat – Handledare – Supervisors		
Antti-Pekka Tuovinen		
Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Pro gradu -tutkielma	03.06.2021	64

Tiivistelmä – Referat – Abstract

Micro frontend -arkkitehtuuri on mikropalveluarkkitehtuurin erikoistapaus, jossa mikropalveluiden rooliin kuuluu tuottaa datan lisäksi myös käyttöliittymä toimintoineen. Toistaiseksi tästä arkkitehtuurimallista on julkaistu kirjallisuutta varsin vähän ja esitetyt mallit ovat enimmäkseen prototyyppitoteutuksia.

Tässä tutkielmassa tarkastellaan erään tietyn ohjelmistoprojektin tuloksena syntynyttä micro frontend -ratkaisua design science -kehyksessä. Aluksi kuvataan toimintaympäristö, ongelmat ja sieltä nousevat vaatimukset. Tähän esitetään Web Components -teknologiaan perustuva ratkaisu, jonka kelpoisuutta arvioidaan sekä sen tuotantokäytöstä saatujen kokemusten perusteella että ATAM-evaluointimenetelmää käyttäen.

Saadut tulokset osoittavat muun muassa sen, että edellä mainittu Web Components -teknologia mahdollistaa HTML-standardin määrittelemien ohjelmointirajapintojen suoran ja tehokkaan käytön web-kehityksessä — ilman tarvetta ohjelmistokehyksille. Lisäksi kyseenalaistetaan näkemys, jonka mukaan micro frontend -arkkitehtuurin kehitys olisi järkevää vain organisatorisista syistä: sille voi olla myös vahvat perustelut esimerkiksi ohjelmiston muokattavuudelle asetettujen vaatimusten vuoksi, ja se on mahdollista toteuttaa myös pienen kehittäjäryhmän toimesta.

ACM Computing Classification System (CCS):
Software and its engineering~Software organization and properties~Software system structures~Software architectures
Computer systems organization~Architectures~Distributed architectures~Client-server architectures

Avainsanat – Nyckelord – Keywords
Micro frontend -arkkitehtuuri, micro-frontend architecture, ohjelmistoarkkitehtuuri, design science

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

1 Johdanto

Micro frontend -arkkitehtuuri on suhteellisen uusi käsite. Se on tunnetumman mikropalveluarkkitehtuurin erikoistapaus, jossa itsenäiset mikropalvelut toimittavat datan lisäksi myös graafisen käyttöliittymän. Micro frontend -arkkitehtuurilla pyritään saamaan mikropalveluiden tuomat hyödyt, mutta siten että niiden integrointi yhdeksi ohjelmistoksi tapahtuu käyttöliittymätasolla.

Mikropalveluarkkitehtuuria (lyhyemmin: mikropalvelut) on käsitelty laajasti kirjallisuudessa [DGL17, EsD16, Fol14, HRI17, Kno16, PMo18, RKU17] ja teeman ympärille on järjestetty lukuisia konferensseja ja seminaareja[1]. Micro frontend -arkkitehtuuria on sen sijaan käsitelty toistaiseksi lähinnä vain teknologiablogeissa ja nettiartikkeleissa, muttei juurikaan tieteellisissä julkaisuissa[2].

Siinä missä mikropalveluilla pilkotaan monoliittisen ohjelmiston liiketoimintakerros pienemmiksi itsenäisiksi palveluiksi, tehdään micro frontend -arkkitehtuurissa sama asia monoliittiselle käyttöliittymälle. Tällainen voi syntyä mikropalveluarkkitehtuuriin siirryttäessä: sen sijaan että monoliitti häviäisi mikropalveluiden myötä, se siirtyykin käyttöliittymätasolle. Esimerkiksi huonekalumyymäläketju IKEA oli todennut tämän ongelman verkkosivustonsa kanssa [IKE18] ja päätynyt omaan micro frontend –ratkaisuun ensimmäisten tunnetuimpien yritysten joukossa.

Samankaltainen tilanne on tämän pro gradu –tutkielman aiheen taustalla. Visma Tampuuri Oy:n Asukassivujen [VIS19] monoliittinen käyttöliittymä oli muodostunut kehityksen pullonkaulaksi, eikä ohjelmisto skaalautunut liiketoiminnan mukana. Tässä tutkielmassa esitellään micro frontend -arkkitehtuuri, joka kehitettiin Asukassivujen uudistuksessa. Näin ollen suunniteltua arkkitehtuurimallia edustaa oikea, tuotantokelpoinen ohjelmisto.

Kirjallisuudessa toistaiseksi käsitellyt micro frontend -tyyppiset ratkaisut [PAS20, HRI17] ovat prototyyppitoteutuksia, joita eivät koske esimerkiksi liiketoimintalähtöiset

[1] Esimerkkeinä https://microservices.sdu.dk/events/, https://apiworld.co/conference/microservices-world/,

[2] Haku Scopus-tutkimustietokantaan ("micro frontend" OR "micro front-end" OR "micro frontends" OR "micro front-ends") tuottaa viisi artikkelia, joista kaksi liittyy tutkielman aiheeseen. Näistä toinen lähteenä tässä tutkielmassa [PAS20]. Lähde: https://www-scopus-com.libproxy.helsinki.fi/search/form.uri?display=basic

tavoitteet ja vaatimukset. Uusien Asukassivujen kohdalla näitä ovat muun muassa:

- helpottaa palvelun käyttöönottoja uusien asiakkaiden kohdalla,
- nopeuttaa uusien ominaisuuksien ja korjausten tuotantojulkaisuja sekä
- vähentää sovellusten kuluttamia resursseja.

Tämän työn tutkimuskysymyksissä tarkastellaan uutta arkkitehtuurimallia ja sen implementaatiota pääasiassa näiden liiketoimintatavoitteiden näkökulmasta.

Ensimmäisessä tutkimuskysymyksessä tarkastellaan,

- TK1: mahdollistaako uusi arkkitehtuuri nopeammat toimitukset.

Nopeammat toimitukset tarkoittavat tässä sekä itse ohjelmiston käyttöönottoa että myös uusien yksittäisten ominaisuuksien tai korjausten julkaisuja.

Tähän liittyy oleellisesti myös asiakaskohtaisista asennuksista luopuminen; uudessa arkkitehtuurimallissa siirrytään moniasiakasmalliin, jossa yhdellä ohjelmistoinstanssilla palvellaan kaikkia asiakkaita. Uuden toteutuksen on pystyttävä säilyttämään asiakaskohtaiset piirteet, kuten käyttöliittymän ulkoasu sekä asiakastilien mukaan vaihteleva sisältö, eli sisäiset tuotesovellukset, joita kutakin vastaa oma mikropalvelunsa. Tästä johdetaan muokattavuuteen ja dynaamisuuteen liittyvä tutkimuskysymys:

- TK2: pystytäänkö instanssi- eli asiakaskohtaisten asennusten räätälöinti korvaamaan moniasiakasmalliin perustuvassa toteutuksessa?

Tämä työ on design science –tyyppinen, joten keskeistä on uuden ratkaisumallin esittäminen olemassa olevaan ongelmaan. Ongelma tämän tutkielman kontekstissa tarkoittaa nykyisten Asukassivujen kohtaamia skaalautuvuuden haasteita. Ongelmaympäristö kuvataan luvussa 2, jossa myös selitettään mikropalveluiden ja monoliitin keskeiset erot. Uusi ratkaisumalli, *design*, kuvataan luvussa 3 sekä suunnitteluprosessina että artefaktina eli syntyneenä ohjelmistoarkkitehtuurina. Luvussa 4 evaluoidaan esitettyä ratkaisua sekä ATAM-metodia käyttäen että tarkastelemalla ohjelman tuotantokäytöstä saatua tietoa. Luvussa 5 pohditaan esitetyn ratkaisun onnistumista ja kehityksessä esiin nousseita asioita. Lopuksi, luvussa 6, esitellään johtopäätökset.

2 Ongelma ja ympäristö

Tässä luvussa selitetään konteksti tutkielman varsinaiselle aiheelle: micro frontend -arkkitehtuuriratkaisulle. Vanhan ohjelmiston arkkitehtuuri ja toteutus korvataan uuden tyyppisellä arkkitehtuurilla. Kyseessä on oikea ohjelmisto, oikeassa toimintaympäristössä, jota siten koskevat myös tietyt erityispiirteet, rajoitukset ja vaatimukset. Tutkielman design science –luonteensa vuoksi on tärkeää kuvata nämä asiat riittävän tarkasti, jotta perustelut esitetyn ratkaisun (Luku 3) taustalla on paremmin ymmärrettävissä. Myös vaihtoehtoisten ratkaisujen pohtiminen on helpompaa, kun pystytään huomioimaan nyt esitettyyn ratkaisuun vaikuttavan ympäristön erityispiirteet.

Tässä tapauksessa myös suunnittelijan taustalla on merkitystä sekä suunnittelun tuloksen että itse tutkielman kannalta. Tekijä tuli ohjelmistoarkkitehdiksi Verkkopalvelu-tiimiin 1.6.2018 yrityksen sisältä vailla kokemusta vanhasta järjestelmästä. Suunnitteluvaiheessa oli siten mahdollista lähteä liikkeelle ilman vanhan ohjelmiston tuomaa ”taakkaa”, ja aiemmin tehtyjä ratkaisuja saattoi kyseenalaistaa. Toisaalta taas joiltain osin ratkaisut saattoivat mennä ylisuunnittelun puolelle, koska selkeää kuvaa kaikista toiminnoista ei ollut. Esimerkiksi tarve useamman itsenäisen mikropalvelun orkestroinnille yhdeksi toiminnoksi ei lopulta realisoitunut tekijän oletuksen mukaisesti.

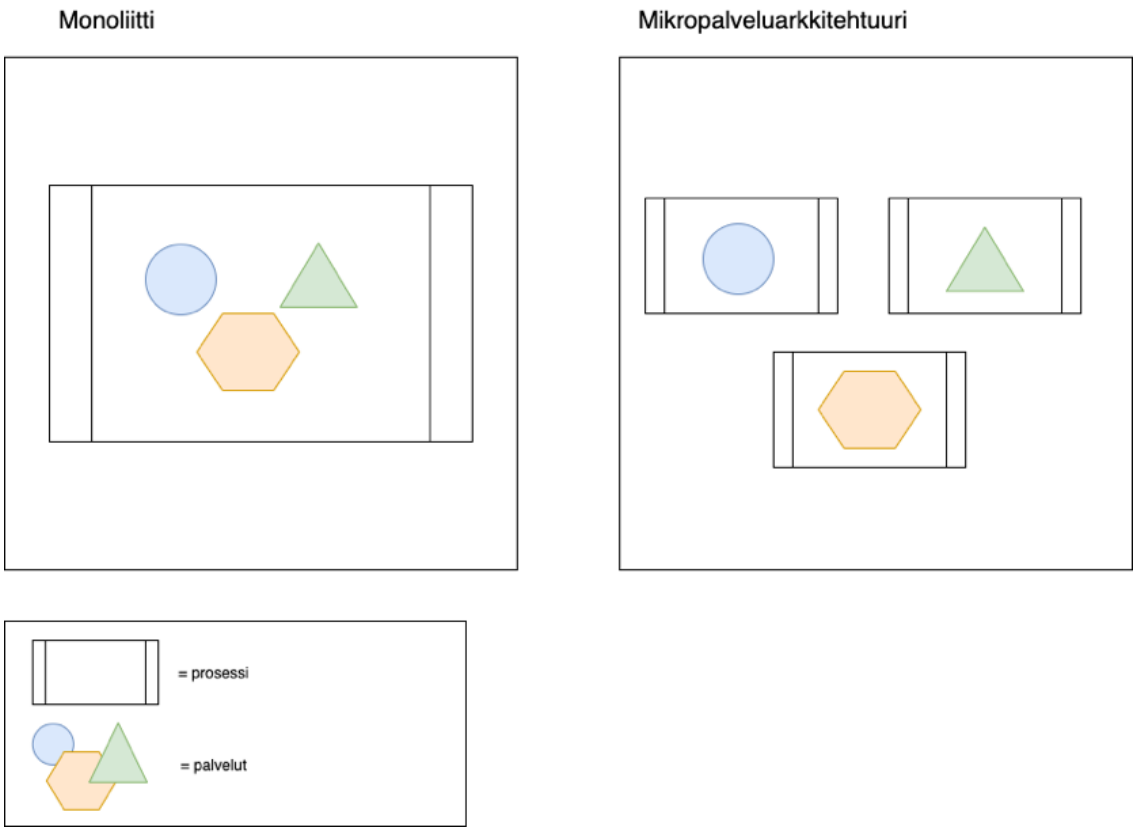
Tämä tutkielma keskittyy micro frontend -arkkitehtuuriin, koska se on aiheena uusi ja tutkielman tekijän päävastuualueena. Toimivan kokonaisuuden kannalta on myös valtava merkitys sillä, kuinka esimerkiksi palvelinpuoli, pilvialusta ja liitännäispalvelut taustalla toimivat. Nämä rajataan kuitenkin tämän tutkielman aiheen ulkopuolelle juuri suunnitteluvastuun mukaan.

Seuraavissa luvuissa käsitellään tämän tutkielman aiheen kontekstia. Aluksi selitetään keskeisten käsitteiden, monoliitin ja mikropalveluiden piirteet. Tapaus Asukassivut -luvussa käydään läpi suurimmat ongelmat vanhassa ohjelmistossa, jonka jälkeen esitellään vaatimukset uudelle ohjelmistoarkkitehtuurille. Näiden jälkeen käsitellään toimintaympäristöä teknisestä näkökulmasta ja käydään läpi web-kehityksen piirteitä. Lopuksi esitellään mikro- ja liitännäispalvelut, jotka osaltaan myös määrittävät uuden ratkaisun toimintaympäristöä.

2.1 Monoliitin ja mikropalveluarkkitehtuurin erot

Monoliitin ja mikropalveluarkkitehtuurin määritelmät perustuvat yksinkertaisimmillaan

ohjelmiston suoritustapaan. Monoliitilla tarkoitetaan ohjelmistoa, jota suoritetaan yhdessä prosessissa, kun taas mikropalveluarkkitehtuuri koostuu joukosta itsenäisiä palveluita, jotka suoritetaan omissa prosesseissaan (Kuva 2.1). Jälkimmäinen mahdollistaa itsenäiset julkaisusyklit kullekin mikropalvelulle [DGL17].



Kuva 2.1. Monoliitti ja mikropalveluarkkitehtuuri.

Monoliitin kohdalla muutosten koodaaminen vaatii useimmiten itse kehitys- ja testausprosessien lisäksi koko ohjelman uudelleen julkaisun. Tämä tarkoittaa sitä, että päivityksen aikana ohjelmisto on poissa käytöstä. Mikropalveluita voidaan sen sijaan kehittää ja päivittää itsenäisesti riippumatta ohjelmiston muista osista.

Mikropalveluarkkitehtuuri on syntynyt ratkaisumalliksi monoliitille, jonka koko on kasvanut sellaiseksi, että sen haitat alkavat olla suurempia kuin hyödyt. Monoliitista mikropalveluarkkitehtuuriin siirtyminen nähdäänkin usein luonnollisena kehityskulkuna ohjelmiston ja sitä kehittävän organisaation tai tiimin koon kasvaessa [EsD16, Kno16].

Mikropalveluarkkitehtuuri tuo mukanaan myös kompleksisuutta. Siinä missä monoliittisessa ohjelmistossa tarvittava tieto ja toiminnot, esimerkiksi apukirjastot, ovat kaiken aikaa saatavilla, on mikropalveluiden kohdalla pyrittävä löytämään vaihtoehtoisia

tapoja toteuttaa ratkaisut. Mikropalveluiden hallinta ohjelmakoodissa voi siten tulla vaatimaan logiikkaa, joka monoliittisessa ohjelmistossa voidaan minimoida tai jopa välttää kokonaan.

Vaikka tämän tutkielmassa ja kirjallisuudessa yleisemmin mikropalvelut nähdään ratkaisuna monoliitin ongelmille, näitä malleja on kyseenalaista verrata keskenään ilman järkevää kontekstia. Voitaneen todeta, että useimmin kyseessä ovat skaalautuvuuden ongelmat. Monoliitille voi olla siis erinomaiset perustelut monissa tapauksissa.

2.2 Tapaus Asukassivut

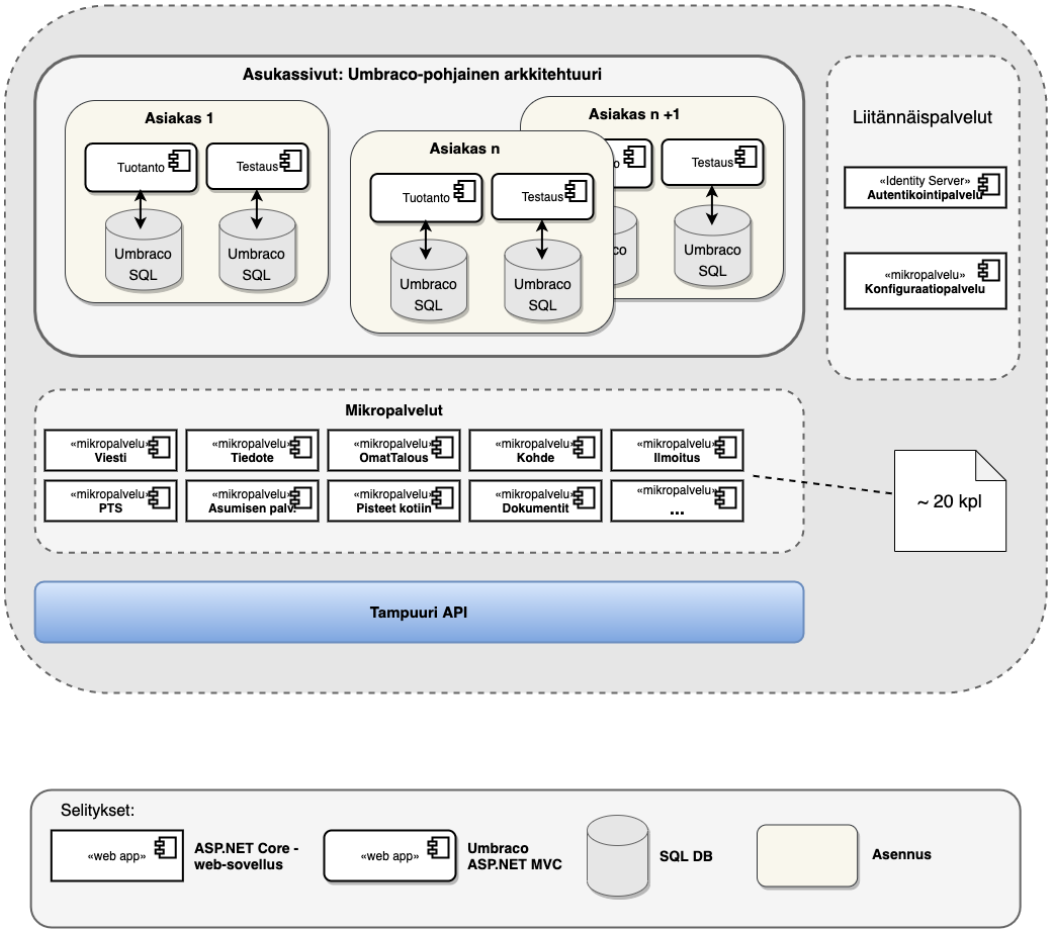
Asukassivut on web-ohjelmisto, joka koostuu erilaisista itsenäisistä tuotteista, joiden avulla vuokrataloyhtiön (asiakasyritys, asiakas) sidosryhmät (loppukäyttäjät) voivat hoitaa asumiseen liittyviä toimintoja, kuten tiedottamista, pesuvuorojen varauksia tai vikailmoitusten tekemistä. Loppukäyttäjiä ovat muun muassa asukkaat, kiinteistönhuoltoyhtiö ja vuokrataloyhtiön edustajat. Käsiteltävä data tulee kiinteistöjenhallinnan toiminnanohjausjärjestelmästä, jota käytetään rajapinnan, Tampuuri APIn, kautta.

Asukassivut ovat palvelleet erinomaisesti tarkoitustaan, joten niille on ollut kysyntää sekä jatkokehitystarpeita. Kysyntään on kuitenkin ollut vaikea vastata, koska ohjelmisto ei skaalaudu riittävästi. Skaalautuminen tässä tarkoittaa sekä itse ohjelmistoa että sen kehittämistä. Tähän vaikuttavat useat seikat.

Umbraco-kehykseen perustuva arkkitehtuuri

Edellisessä luvussa selitetty monoliitin ongelma realisoituu Asukassivujen kohdalla käyttöliittymätasolla. Alun perin Asukassivujen käytön otaksuttiin painottuvan vahvasti sisällönhallintaan. Ohjelmisto rakennettiin ASP.NET MVC -kehykseen pohjautuvaa Umbraco[3]-sisällönhallintajärjestelmää käyttäen (Kuva 2.2).

[3] Umbraco: https://umbraco.com/



Kuva 2.2. Asukassivujen kokonaisarkkitehtuuri. Yläosassa Asukassivut-ohjelmiston käyttöliittymäsovelluksen muodostava osuus: Umbraco-kehykseen pohjautuva monoliittinen arkkitehtuurimalli. Jokaisella asiakkaalla on sekä tuotanto- että testausympäristöt.

Huolimatta monista hyödyistä, joita erikoistunut Umbraco-kehys tuotti (esimerkiksi sivustorakenne, käyttäjäroolien hallinta), se kuitenkin alkoi muuttuvien vaatimusten ja vaihtelevien koodauskäytäntöjen myötä tuottaa ongelmia kehitykselle ja käyttöönotolle. Sisällönhallintaominaisuudet alkoivat jäädä toissijaiseksi ja Asukassivuja alettiin kehittää monipuolisemmaksi viestintä- ja informaatiojärjestelmäksi.

Asiakasmäärien kasvaessa myös asiakaskohtaiset konfiguroinnit alkoivat muodostua vaikeammin hallittaviksi. Näillä tarkoitetaan sivuston ulkoasuun liittyviä asioita, kuten logoja ja värejä, sekä myös asiakkaan toimialasta (vuokrataloyhtiö tai isännöinti) riippuvia asioita, kuten sivustolla näytettävät tuotteet.

Käyttöönotot alkoivat olla selvittelyineen työläitä ja aikaa vieviä. Konfiguraatioidenhallinta ei ole ollut kovin organisoitua. Tavat toteuttaa konfiguraatio on saattanut vaihdella tekijänsä mukaan: osa konfiguroinneista määritelty ohjelmiston konfigurointitiedostoon, osa taas Umbracon käyttämään SQL-tietokantaan. Näiden selvittely ja hallinta kuormittivat runsaasti sovelluskehittäjiä, joten ne söivät samalla

uuteen kehitykseen käytettävää aikaa. Myös Umbracosta itsestään alkoi tulla myös kehityksen este: sen kehittäjien lupaamaa ASP.NET Core -kehystä tukevaa versiota ei kuulunut lupauksista huolimatta, joten se käytännössä esti osaltaan Asukassivujen modernisoinnin.

Arkkitehtuuriuudistuksen myötä olisi myös tarkoitus siirtyä asiakaskohtaisista asennuksista niin kutsuttuun moniasiakasmalliin (*multi tenancy*). Tämä ratkaisu poistaisi ohjelmiston asiakaskohtaiset instanssit, jotka jokaisen uuden asiakkaan myötä kasvattavat sovellusinfraa lineaarisesti: kullekin uudelle asiakkaalle pystytetään testaus- ja tuotantoympäristöt, mikä tarkoittaa Azuren[4] pilvialustalla kahta Web app -palvelua ja kahta uutta tietokantainstanssia (Kuva 2.2).

Moniasiakasmallilla olisi positiivinen vaikutus ylläpidon kuormaan. Lisäksi se yksinkertaistaisi myös julkaisuprosessia ja mahdollistaisi jatkossa käyttöönoton ilman teknistä osaamista. Olemassa olevat mikropalvelut ovat valmiiksi moniasiakasmallia tukevia, joten tältä osin muutos koskisi ainoastaan monoliittia (Kuvassa 2.2. ”Umbraco-pohjainen arkkitehtuuri”).

Mikropalveluiden periaatteisiin liittyy oleellisesti omat itsenäiset julkaisusyklit, teknologinen riippumattomuus toisista mikropalveluista ja näitä kehittävistä itsenäisistä tiimeistä [FoL14, DGL17]. Asukassivujen tapauksessa ohjelmistoa kehittää kuitenkin yksi, noin 5-7 hengen tiimi. Tämä on mittaluokaltaan pieni verrattuna ohjelmistoihin, joissa puhutaan useamman tiimin kehityksestä. Tämän vuoksi on mahdollista linjata yhteisiä toimintamalleja ja sopia käytetyistä teknologioista. Asukassivujen kehitys ei ole, eikä luultavasti jatkossakaan ole, jaettu eri tiimien välille, joten juuri nuo teknologiset ratkaisut ovat oman tiimin arkkitehtien päätäntävallassa.

2.3 Laadulliset vaatimukset

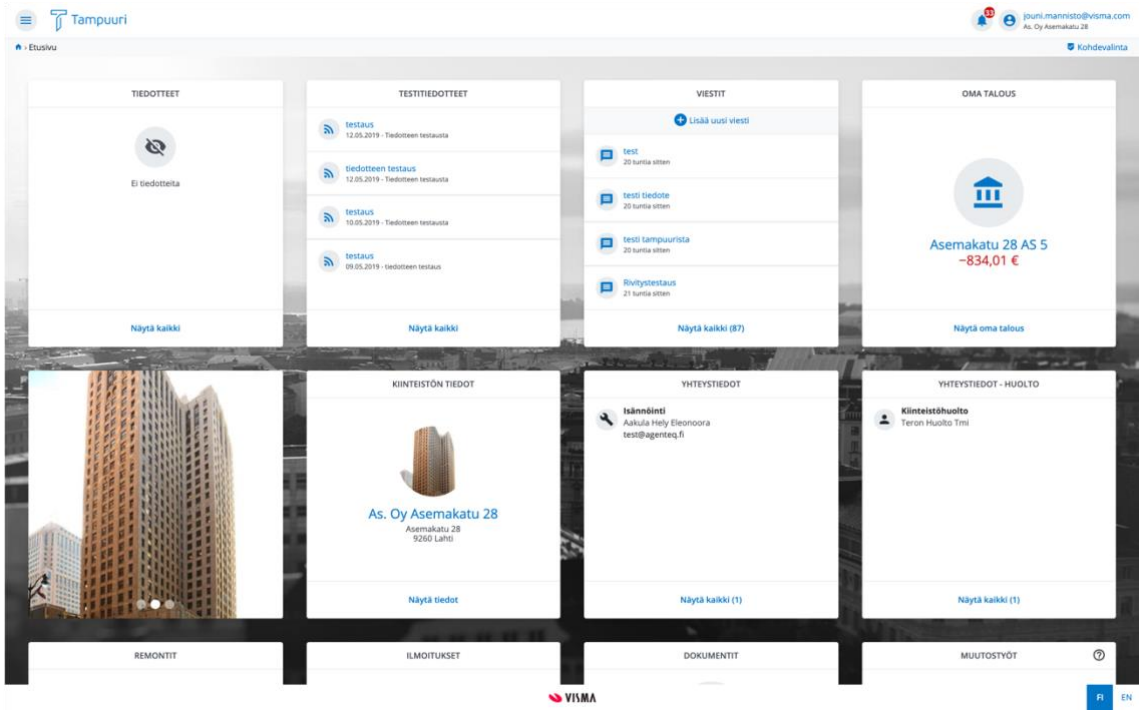
Uusien Asukassivujen toiminnalliset vaatimukset periytyvät suoraan vanhalta ohjelmistolta. Ohjelmiston liiketoimintalogiikka on jo vanhassa toteutuksessa jaettu omiin mikropalveluihinsa, joten muutokset kohdistuvat tapaan toteuttaa käyttöliittymäkerros jo olemassa oleville toiminnoille. Tästä syystä yksittäisiä käyttötapauksia ei tässä tutkielmassa ole syytä käsitellä.

---

[4] Microsoft Azure https://docs.microsoft.com/fi-fi/azure/?product=featured ja App Service https://docs.microsoft.com/en-us/azure/app-service/overview

Käyttöliittymämalli

Vanhojen Asukassivujen käyttöliittymä perustuu pieniin kortteihin, ”tiiliin”, jotka edustavat aina jonkin liiketoiminta-alueen toiminnallisuutta ja joissa on yhteenveto sillä hetkellä käyttäjän kannalta relevanteista asioista. Etusivulta nähdään yhdellä silmäyksellä esimerkiksi uusimmat viestit tai vikailmoitukset (Kuva 2.3). Kukin tiili toimii siis sekä yhteenvetona että myös linkkinä varsinaiselle tuotteelle, jossa kyseisiä tietoja voidaan käsitellä laajemmin. Tämä tiiliin pohjautuva käyttöliittymämalli oli se, jota myös uuden ohjelmiston tulisi noudattaa.



Kuva 2.3. Asukassivut-tuotteen aloitussivu. Jokainen ”tiili” näyttää yhteenvedon edustamastaan datasta ja toimii linkkinä joko tarkempiin tietoihin tai sisältää oikopolun johonkin toimintoon, esimerkiksi uuden viestin (ylärivi, toinen oikealta) luomiseen. Kuva on Asukassivujen uudesta toteutuksesta.

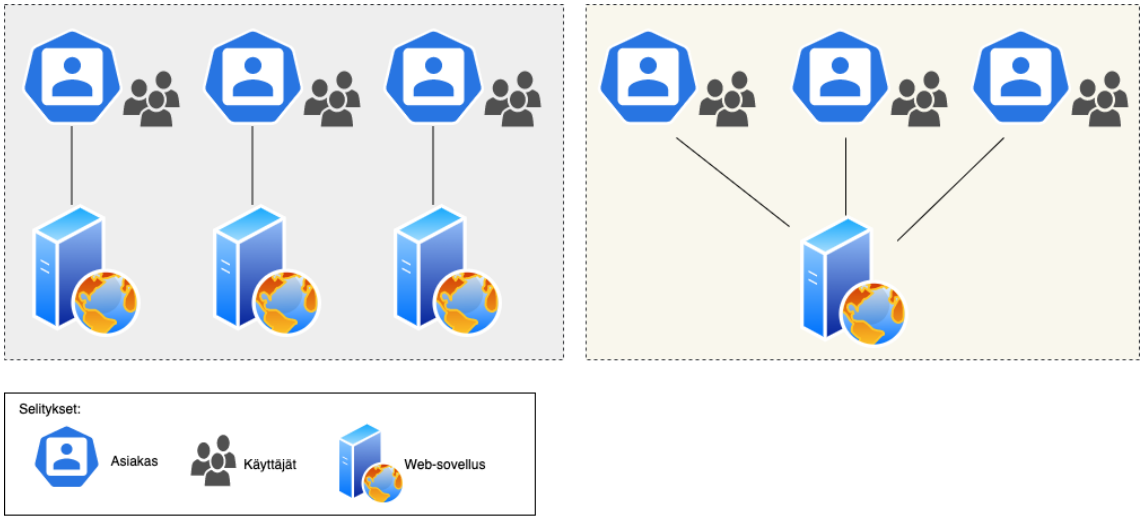
”Tiili” terminä kuvaa vanhojen Asukassivujen kohdalla sekä käyttöliittymäelementtiä että itse ominaisuutta. Alkuvaiheessa suunnittelussa otettiin käyttöön käsite ”tuote” kuvaamaan täsmällisemmin reaalimaailman tilannetta: asiakkaalla on käytössä tietyt tuotteet ja tuotteella tulee olemaan erilaisia näkymiä, muun muassa tiiliä. Jatkossa tekstissä ”tuote” tai ”tuotepalvelu” tarkoittaa erillistä micro frontend -palvelua, joka tuottaa datan näkymineen Asukassivut-ohjelmistolle esimerkiksi Kuvan 2.3 tiilien mukaisesti.

Tärkeimmät laadulliset vaatimukset nousivat esille vanhojen Asukassivujen ongelmista. Uusien toiminnallisuuksien kehittäminen ja korjausten saaminen tuotantoon oli hidasta

johtuen muun muassa monoliittisesta käyttöliittymästä. Myöskin käyttöönottoprosessia olisi tärkeä saada jouhevammaksi ja vähemmän kehittäjien aikaa vieväksi. Käyttöönotto olisi jatkossa oman erillisen tiimin vastuualueella, joten tämä tulisi vaikuttamaan suunnitteluun: jatkossa asiakaskohtaiset konfiguroinnit vietäisiin erilliseen konfiguraatiopalveluun. Tämä oli jo ollut aiemmin suunnitteluissa mukana (näkyy myös Kuvassa 2.2), mutta sitä ei oltu vielä käytetty. Asukassivujen myötä sen on tarkoitus olla keskeisessä roolissa muokattavuudenhallinnassa.

Muokattavuus moniasiakasympäristössä

Eräs keskeisin huomioitava asia uudessa arkkitehtuurissa on siirtyminen yksiasiakasmallista moniasiakasmalliin (Kuva 2.4). Vanhat Asukassivut toimivat siten, että kullakin asiakkaalla on oma instanssi ohjelmistosta. Tällainen ”instanssi-per-asiakas”-malli (Kuva 2.4, vasen puoli) mahdollistaa vaihtelevat, asiakaskohtaiset piirteet ohjelmistossa, mutta aiheuttaa vastaavasti ylläpidollisia haasteita ja kustannuksia asiakasympäristöjen lukumäärän kasvaessa. Moniasiakasmallissa (oikea puoli) kaikki asiakkaat jakavat web-ohjelmiston saman instanssin. Asukassivujen mikropalvelut tukevat moniasiakasmallia valmiiksi.



Kuva 2.4. Vasemmalla instanssi per asiakas- eli yksiasiakasmalli (*single-tenant*), oikealla monta asiakasta per instanssi- eli moniasiakasmalli (*multi-tenant*).

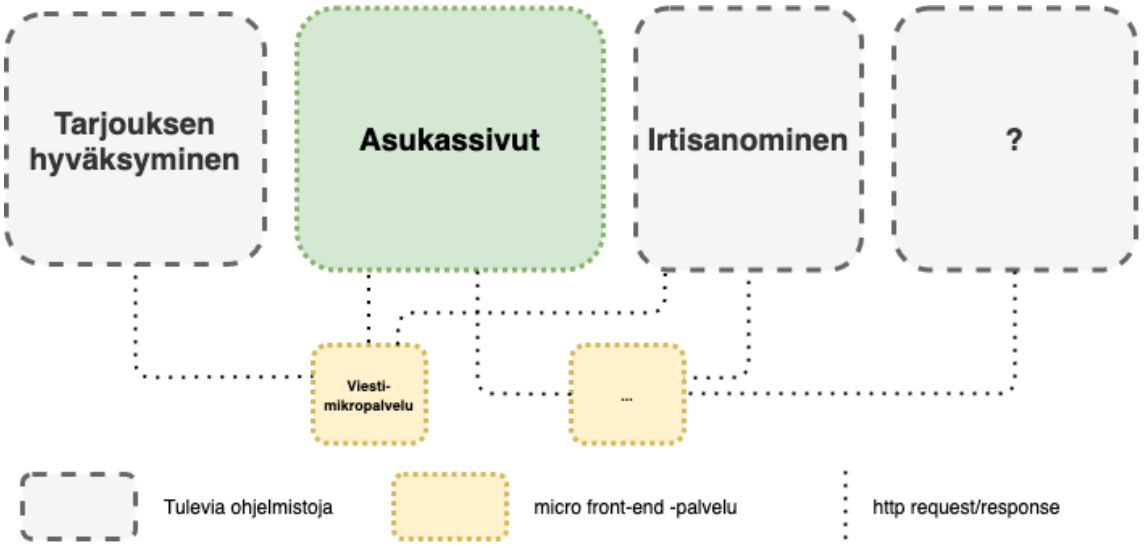
Käytännössä ohjelmiston on siis muututtava dynaamisesti sekä ulkoasultaan että tuotekokoonpanoltaan kirjautuneen käyttäjän asiakastilin mukaan. Tämä vaatimus käyttöliittymälle on yksi merkittävistä eroista verrattuna tällä hetkellä tieteellisessä kirjallisuudessa [PAS20, HRI17] ja Internet-lähteissä [IKE18, Zal18, JaC19, Gee19]

esitettyihin ratkaisuihin. Useimmiten näissä käsitellään staattisina pysyvää mikropalvelujen joukkoa, tyyliin verkkokauppa.

Esitetyt tutkimuskysymykset liittyvät juuri edellä esitettyihin asioihin: saadaanko uudesta moniasiakasmallia tukevasta ratkaisusta sellainen, joka mahdollistaa nopeammat käyttöönotot (TK1), ja pystytäänkö tällaisella mallilla vastaamaan muokattavuuden haasteisiin riittävällä tasolla (TK2)?

Palveluiden uudelleenkäytettävyys

Ominaisuus jota ei juurikaan ole käsitelty artikkeleissa tai teknologiablokeissa, liittyy micro frontend -palveluiden uudelleenkäyttöön. Esimerkiksi eri sidosryhmien väliseen viestintään kehitettyä Viesti-tuotetta voidaan jatkossa käyttää muuallakin kuin vain Asukassivuilla. Ennen kuin asunnonhakijasta tulee asukas, hän voi kommunikoida vuokrataloyhtiön kanssa asunnon hakemiseen tai asunnon vastaanottamiseen liittyvissä asioissa (Kuva 2.5). Tämä ei ole Asukassivujen toteutuksen kannalta ensimmäisessä vaiheessa kriittinen vaatimus, mutta jo suunnitteluvaiheessa tämä näkökulma oli mukana.



Kuva 2.5. Micro frontend -palveluiden uudelleenkäyttö. Asukassivujen rinnalle kehitettävät ohjelmistot noudattavat Asukassivujen micro frontend -arkkitehtuuria. Tällöin kokonaisuus mahdollistaa myös mikropalvelutasoisen uudelleenkäytön eli sen, että yhdellä mikropalvelulla on useampi sitä käyttävä ohjelmisto.

2.4 Teknologinen kehys

Jokaisessa toimintaympäristössä on omat vaatimuksensa ja rajoitteensa, jotka asettavat raamit ja liikkumavaran teknologioiden suhteen. Vaikka Asukassivujen arkkitehtuurin uudistaminen on yhden tiimin ja yrityksenkin mittakaavassa suurehko hanke, ei siinäkään uudisteta kaikkea. Esimerkiksi data haetaan edelleen mikropalveluista ja käyttäjän tunnistautuminen tapahtuu olemassa olevalla autentikointipalvelimella. Ja vaikka asiakaspuolen (tässä: asiakas-palvelin-konteksti) koodipohja uudistuu, palvelinpuolella ohjelmointikieli ja ohjelmistokehys pysyvät versionostoja ja -päivityksiä lukuun ottamatta samoina. Suurimmat uudistukset koskevat siis asiakaspään ratkaisuja.

Vanhat Asukassivut pohjautuvat ASP.NET-ohjelmistokehykseen, jonka dynaamisuus perustuu palvelinpuolen renderöintiin (server-side rendering, SSR). Tällöin käyttäjän selaimelle lähetettävä HTML-sivu muodostetaan palvelimella ja lähetetään valmiina dokumenttina selaimeen. Kehitysaikana HTML-sivut koodataan käyttäen ohjelmistokehyksen tarjoamaa skriptikieltä (ASP.NET Razor), jonka avulla HTML-sivun dynaaminen osa voidaan muodostaa käyttäen C#-kieltä ja sen rakenteita kuten silmukoita ja ehtolauseita.

Usein mikropalveluiden puolesta argumentoidessa pidetään yhtenä oleellisista seikoista mahdollisuus valita teknologiat mikropalveluiden — ja myös tiimien — välillä [DGL17, RKU17]. Tämä pätee myös micro frontend -arkkitehtuuriin, mutta on kuitenkin huomioitava, että verkkoa tulee kuormittamaan datan lisäksi myös käyttöliittymän muodostava koodi. Eli pelkästään mahdollisuus käyttää useita eri teknologioita ei voi olla itse tarkoitus.

ASP.NET Core

Kuten kokonaisarkkitehtuurin kuvauksesta käy ilmi (Kuva 2.2), kaikki muut palvelut paitsi Umbraco-osuus on rakennettu ASP.NET Core -ohjelmistokehystä [Mic21] hyödyntäen. Sillä voidaan toteuttaa web-sovelluksia, jotka sisältävät sekä asiakas- ja palvelinpään toteutuksen tai vain jälkimmäisen. Tätä tapaa edustavat käytännössä kaikki mikro- ja liitännäispalvelut. Kommunikaatio kokonaisarkkitehtuurissa perustuu näiden tarjoamille REST-rajapinnoille. Tulevassa ratkaisussa hyödynnetään samaa ohjelmistokehystä riippumatta asiakaspään toteutustavasta.

Web-kehityksen piirteitä: kehykset ja arkkitehtuurimallit

Web-pohjaisessa sovelluskehityksessä kaikelle käyttöliittymätason tekemiselle asettaa raamit HTML:n standardit. Micro frontendin tapauksessa HTML-standardin document object model (DOM) on tärkein ohjelmointirajapinta. Sitä käyttäen mikropalvelujen tuottamat näkymät koostetaan yhdeksi kokonaisuudeksi.

Eri selainvalmistajat käyttävät omia JavaScript-moottoreitaan [JS1] ja implementoivat standardiin hyväksyttyjä ominaisuuksia eri tahtiin. Tästä syystä JavaScript-kieleen hyväksytty ominaisuus ei välttämättä ole toiminnassa heti seuraavassa selaimen julkaisuversiossa. Muun muassa tästä syystä erilaisille apukirjastoille on ollut kysyntää kehittäjien keskuudessa. Käytetyin näistä yhä edelleen on jQuery[5].

Lukuisia ohjelmistokehyksiä[6] on kehitetty helpottamaan koodin organisoimisessa erilaisten arkkitehtuurimallien mukaisesti. Erityisesti *model-view-controller*-malli (MVC) ja sen variaatiot (MVVM, MV*) ovat olleet useiden eri ohjelmistokehysten perustana. Tällaisia ovat muun muassa Backbone (MVC) ja Knockout (MVVM). Reactin myötä komponenttipohjaisuus on levinnyt laajasti ja on tällä hetkellä suosituimpien JavaScript-kehyksien perustana. Komponenttipohjaisuus mahdollistaa koodin uudelleenkäytön sekä sovelluksen sisällä että myös sovellusten välillä. Uudelleenkäytettävyys tuo muutakin hyötyä kuin vain nopeuden. Virheettömyys ja pienempi alttius haavoittuvuuksille on myös nähty uudelleenkäytön tuomina etuina. Näitä on perusteltu muun muassa sillä, että komponenttia käyttää useampi kehittäjä [OJK19].

Kehykset tarjoavat usein valmiit ratkaisut erilaisiin web-kehityksessä tyypillisiin ongelmiin ja tilanteisiin: esimerkiksi sivustolla tapahtuvaan navigointiin tai sovelluksen tilanhallintaan. Lisäksi kehysten mukana tulevat tarvittavat käytännöt, työkalut ja konfiguraatiot, joita etenkin web-kehityksessä toistaiseksi tarvitaan, ennen kuin koodi on

---

[5] Vuonna 2019 yli 75% 10 miljoonasta suosituimmasta web-sivustosta käytti jQuery-kirjastoa. Lähde: https://w3techs.com/technologies/overview/javascript_library

[6] Käsitteitä ”kirjasto” ja ”ohjelmistokehys” käytetään erityisesti web-kehityksessä sekaisin. Esimerkiksi: ”React - A JavaScript *library* for building user interfaces”, kun taas lukuisat listaukset taas käyttävät Reactista termiä ”ohjelmistokehys” Lähteet: https://reactjs.org/ https://2019.stateofjs.com/front-end-frameworks/

selaimen suoritettavissa[7].

Aiemmin mainittu selainten kirjo on yksi syy tähän valtavaan työkalujen tarpeeseen. Modernit web-selaimet ja näiden uusimmat versiot tukevat HTML-standardin ja JavaScriptin uusia ominaisuuksia, mutta vanhoja selaimia voi olla käytössä suurella joukolla loppukäyttäjistä. Yksi toimintamalli on muun muassa sellainen, jossa eri selaimille — karkeasti jaotellen: modernit ja *legacy*-selaimet — tuotetaan erilaiset käännökset, *buildit*, samasta lähdekoodista.

Moni seikka puhuu erilaisten ohjelmistokirjastojen ja -kehysten käytön puolesta. Edellä mainittujen lisäksi muun muassa laaja ja vuosia kestänyt tuotantokäyttö kertoo toimintavarmuudesta ja pienentää mahdollisuutta törmätä ongelmiin, joita ei olisi jo ehditty ratkaista aikaisemmin. Suuri ja aktiivinen kehittäjäyhteisö nopeuttaa ongelmatilanteissa ratkaisun löytymistä, ja todennäköisesti myös dokumentaatio on riittävän laadukasta tukemaan kehitystä.

Kehyksissä on myös kääntöpuolensa. Riippuvuus johonkin ulkopuolisen tuottamaan koodiin voi muodostaa riskin tulevaisuudessa. Onko esimerkiksi takeita siitä, että jotain tiettyä kirjastoa ylläpidetään jatkossa ja tarvittavat tietoturvapäivitykset tulevat ajallaan? Ja päteekö tämä kirjaston omien riippuvuuksien kohdalla? Onko kehittäjäyhteisöllä resursseja korjata virheitä ja kehittää sitä jatkossa standardien kehittyessä?

Vanhojen Asukassivujen tapauksessa käytännön esimerkki liittyi Umbracoon eli sisällönhallintajärjestelmän ohjelmistokehykseen. Sitä kehittävä yhteisö oli luvannut tietyllä aikajänteellä julkaista version, joka olisi yhteensopiva ASP.NET Core -kehyksen kanssa, mutta julkaisua ei kuulunut useaan vuoteen[8]. Pahimmillaan ulkopuolinen riippuvuus voi viivästyttää tai haitata sovellusympäristön kannalta oleellisia versionostoja tai tietoturvapäivityksiä. Kaiken kaikkiaan mahdollisten ohjelmistokehysten käyttö vaatii pohdintaa eri näkökulmista, koska sekä puolesta että vastaan löytyy varsin hyviä perusteluja.

---

[7] Ns. modernissa web-kehityksessä JavaScript-koodi tyypillisesti käännetään taaksepäin yhteensopivaksi eli ”transpiloidaan”, minifioidaan ja ”bundlataan” (eli niputetaan se sopivan kokoisiksi tiedostoiksi), ennen kuin se on selaimen ladattavissa ja suoritettavissa. Jälkimmäiset työvaiheet tehdään yleensä tuotantojulkaisuissa.

[8] Syyskuussa 2020 julkaistu blogi Core-version alpha-versiosta: https://umbraco.com/blog/net-core-alpha-release/

2.5 Mikropalvelut ja liitännäispalvelut

Sekä vanhat että uudet sivut käyttävät toiminnoissaan erillisiä palveluita. Autentikointipalvelin hoitaa käyttäjän tunnistautumisen, ja mikropalvelut sisältävät vastuualueidensa liiketoimintalogiikan. Konfiguraatiopalvelu tulee olemaan keskeisessä roolissa siirryttäessä moniasiakasmalliin. Asiakaskohtaiset konfiguraatiot hoidetaan jatkossa sen kautta, ja käyttöönottoprosessia pyritään helpottamaan konfiguraatiopalvelua varten rakennettavalla konfigurointityökalulla. Graafinen käyttöliittymä mahdollistaisi käyttöönottojen suorittamisen myös muille kuin sovelluskehittäjille. (Tämän varsinainen implementointi tulisi sitä käyttävän käyttöönottotiimin vastuulle.)

3 Micro frontend -arkkitehtuuri

”Suunnitteluprosessi on asiantuntijatyötä, jonka tuloksena syntyy innovatiivinen artefakti” [HMS04]. Design science -kontekstissa suunnittelu, *design*, tarkoittaa siis sekä itse prosessia että sen lopputulosta. Tässä luvussa kuvataan molemmat. Tutkielman aihe on reaalimaailman tapaus, joten on perusteltua tuoda esille pohdintaa suunnittelupäätösten taustalla. Kronologisesti etenevä kuvaus avaa parhaiten perusteluja päätösten taustalla.

Uudessa arkkitehtuurissa siirryttiin moniasiakasmalliin, joten se itsessään on jo arkkitehtuurillinen päätös. Samoin siihen liittyvä konfiguraatiopalvelun roolin kasvattaminen (Taulukko 3.1). Nämä olivat jo ennalta tehtyjä linjauksia, jotka olivat ohjaavina tekijöinä tuleville päätöksille.

Linjaus	**Selitys**
L1	Moniasiakasmalliin siirtyminen (Kuva 2.3).
L2	Konfiguraatiopalvelun roolin kasvattaminen

Taulukko 3.1. Suunnittelua ohjaavat linjaukset.

Aliluvussa 3.1 kuvataan suunnittelun eteneminen ja syntyneet suunnittelupäätökset pohdintoineen ja perusteluineen. Tämän jälkeen, aliluvussa 3.2 kuvataan tulos: kokonaisarkkitehtuuri, jossa Umbraco-pohjainen arkkitehtuuri on korvattu micro frontend -arkkitehtuurilla, sekä itse micro frontendin oleelliset piirteet.

3.1 Suunnitteluprosessi ja -päätökset

Micro frontend -arkkitehtuurin suunnittelussa lähdettiin tekemään prototyyppejä ja *proof of concept* -toteutuksia (POC), joiden pohjalta edettiin kohti seuraavia vaiheita. Pohjateknologian valinta oli uusien Asukassivujen kannalta kaikkein oleellisin päätös. Eli piti päättää se teknologia ja tapa, jolla eri näkymät toteutetaan ja kuinka ne integroidaan yhdeksi kokonaisuudeksi. Vaihtoehtoina olisi joko 1.) hyödyntää HTML-standardia mahdollisimman pitkälle ja kirjoittaa koodi itse, 2.) hallita näkymiä jollain JavaScript-kirjastolla tai 3.) näiden molempien kombinaatio. Kaksi jälkimmäistä tarkoittaisi sitä, että toteutus liittäisi valitun kirjaston kiinteästi osaksi ratkaisua ja ohjelmiston ylläpito olisi jatkossa kiinteästi osa tuon kirjaston elinkaarta.

Suunnittelupäätökset 1 ja 2 (SP1, SP2): SPA ja CSR

Uusien Asukassivujen sisältö tulisi olemaan hyvin dynaamista, joten palvelinpään renderöinti tuntui lähtökohtaisesti joustamattomalta mallilta. Ainakin lähtökohtana toteutukselle tulisi olemaan niin sanottu *single-page application*[9]- toteutus (SPA), joka perustuisi vahvasti asiakaspään renderöintiin (*client-side rendering*, CSR). Kommunikointi asiakaspäästä palvelimelle toimisi jatkossa ajax-kutsuin REST-rajapintaa käyttäen ja sivuston sisältö muodostettaisiin selaimessa.

Nämä ensimmäiset suunnittelupäätökset (SP1 ja SP2) veisivät kehitystä modernimman web-kehityksen suuntaan, jolla pyritään muun muassa parempaan käyttäjäkokemukseen. SPA-toteutuksella sivuston staattiset osat voidaan renderöidä nopeasti ja vain vaihtuva sisältö haetaan palvelimelta.

Teknologiaselvitykset

Uudistuksen alkuvaiheessa esille nousi kaksi kiinnostavaa projektia, jotka olivat jakaneet kokemuksiaan käyttöliittymätason ratkaisuistaan mikropalveluarkkitehtuurissa. IKEAn ja Zalandon [Zal18, IKE18, PMo18] verkkokaupat olivat siirtyneet micro frontend -arkkitehtuuriin. Näiden verkkokauppojen käyttäjämäärät ja käytettävät resurssit ovat eri luokkaa kuin Verkkopalvelu-tiimin ja Asukassivujen vastaavat, mutta teknologian ollessa verrattain uusi, on mahdollista saada hyödyllistä tietoa muiden kokemuksista. Joka tapauksessa konteksti on kaikissa sama: web-kehitys – vaikka erojakin on paljon.

Näiden yritysten teknologiablogien myötä erilaisia mahdollisuuksia nousi esille ja päätyi siten lähempään tarkasteluun. Osa oli jo tuossa vaiheessa jossain määrin vanhentunutta ja osa vasta nousemassa esille, laajempaan tietoisuuteen.

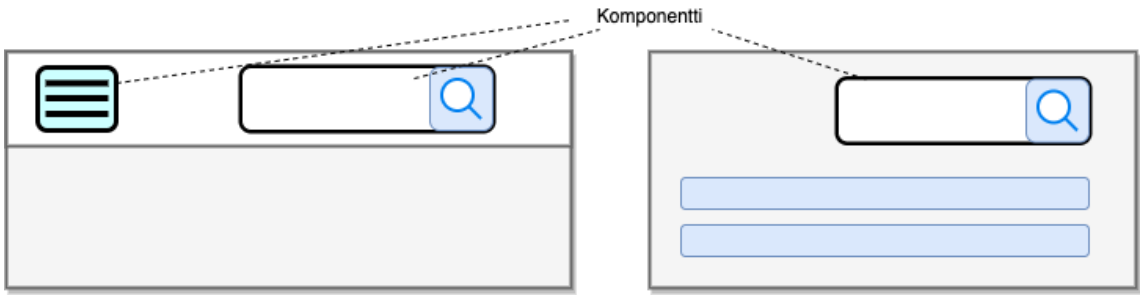
Jälkimmäistä edustaa Web Components. Se tarkoittaa joukkoa erilaisia teknologioita, joiden avulla kehittäjä voi muun muassa rakentaa uudelleenkäytettäviä käyttöliittymäkomponentteja. Nimensä mukaisesti se tuo komponenttipohjaisuuden osaksi HTML-standardia. Vaikka näiden standardien implementaatio eri selainvalmistajien välillä oli vielä kesken, oli se kuitenkin jo selkeästi etenemässä.

---

[9] *Single-page application* (SPA) tarkoittaa sovellusta, jossa käyttöliittymä on toteutettu yhdellä html-sivulla ja jonka sisältöä päivitetään osissa lataamatta itse sivua uudelleen. Vertaa *multi-page application*, jossa sisältö esitetään useammalla html-sivulla (esim. etusivu.html, yhteystiedot.html) ja näiden välillä navigoidutaan hyperlinkein.

Käytännössä uudelleenkäytettävien komponenttien rakentaminen tapahtuu Web Components -standardiin kuuluvilla Custom Elements -ohjelmointirajapinnoilla [MDN20d]. Jatkossa termillä ”web-komponentti” tarkoitetaan tekstissä *yksinomaan* tällaista Custom Elements -komponenttia. Web-komponentit vaikuttivat olevan ikään kuin standardoituja versioita esimerkiksi React-komponenteista [REA20a].

React on yksi tämän hetken käytetyimpiä JavaScript-kirjastoja. Se perustuu erityisesti uudelleenkäytettäviin komponentteihin. Esimerkiksi käyttöliittymän otsikkokomponentti voi sisältää navigaatio- ja hakukomponentin (Kuva 3.1). Samaa hakukomponenttia voidaan käyttää toisaalla ja samoin otsikkokomponentti voi toistua useammalla HTML-sivulla. Näin koko sovellus koostuu erikokoisista komponenteista ja yhden komponentin koodi on kirjoitettu vain kerran.



Kuva 3.1. Geneerinen esimerkki komponenttipohjaisuudesta käyttöliittymätasolla: otsikkokomponentti, joka koostuu sekä navigaatio- että hakukomponentista. Koko sovellus on kokoelma komponentteja, joilla on vastuu yhden asian tekemisestä.

React oli hyvä vertailukohdaksi muun muassa kehittäjäkokemuksensa vuoksi. Uudelleenkäytettävät komponentit ovat Reactin ideologian ydin. Ja komponentteina myös mikropalveluiden käyttöliittymät olisi järkevä miettiä; tuolloin esimerkiksi niiden näkyvyyden hallinta ohjelmallisesti JavaScriptillä olisi triviaalia.

React on kirjastona esimerkillinen myös testauskäytäntöjen suhteen. Ensinnäkin, jo edellä mainittu komponenttipohjaisuus pakottaa jaottelemaan sovelluksen osiin, jolloin näiden testaamisesta tulee helpompaa ja toiseksi, valmiit, hyvin yhteensopivat testaustyökalut ja -käytännöt tekevät asiakaspään koodin testauksesta helppoa. Jos jonkin kirjaston tai kehyksen katsottaisiin olevan hyödyllinen micro frontendiä implementoitaessa, olisi React yksi mahdollinen kandidaatti.

Suunnittelupäätös 3 (SP3): komponenttipohjaisuus web-komponentein

Komponenttipohjaisuuteen liittyy myös komponenttien uudelleenkäytön lisäksi niiden jakaminen: sekä sovelluksen sisällä että tiimien välillä. Toki myös avoimena lähdekoodina. Reactin kohdalla yksi kehittäjäkokemusta parantava piirre on sen yleisyys ja sen mukanaan tuomat toisten kehittäjien jakamat komponentit. Esimerkiksi päivämäärän valintaan tarvittavasta käyttöliittymäkomponentista löytyy useita käyttökelpoisia React-komponentteja, joten kehittäjän ei tarvitse käyttää aikaa sellaisen, tai vastaavien muiden komponenttien koodaamiseen.

Aiemmin todetut yhtäläisyydet web- ja React-komponenttien välillä oli syytä tutkia tarkemmin. Jos komponenttien hyödyt saataisiin käyttöön ilman sidosta johonkin tiettyyn ohjelmistokirjastoon tai -kehykseen, olisi se etenkin micro frontend -implementaation ollessa kyseessä hyvä vaihtokauppa: yksi tavoitteista olisi säilyttää teknologinen riippumattomuus mikropalveluiden välillä. Tämän tulisi päteä myös käyttöliittymäratkaisussa.

Natiivit web-komponentit ovat ohjelmoijan määrittelemiä DOM-elementtejä, joilla on oma sisäinen toiminnallisuutensa, ja jotka selain osaa renderöidä tavallisten HTML-elementtien tavoin. Sisäinen toiminnallisuus tarkoittaa logiikkaa, jolla komponentti reagoi sen elinkaarifunktioihin tai arvojen muutoksiin (elementin attribuutit DOMissa tai kentät JavaScript-koodissa).

Elinkaarifunktiot ovat web-komponenttien spesifikaatiossa määritellyt funktiot, jotka suoritetaan aina tietyssä vaiheessa kyseisen komponentin elinkaarta. Esimerkiksi *connectedCallback*-funktio suoritetaan, kun elementti liitetään DOMiin ja vastaavasti *disconnectedCallback* suoritetaan elementin poistuessa DOMista.

Attribuutit tarkoittavat samaa kuin millä tahansa HTML-elementillä: käyttävä taho ohjaa elementtiä attribuuteilla, joihin se reagoi oman sisäisen logiikkansa mukaisesti. Attribuuttien voidaan ajatella olevan kehittäjän julkaisemaa rajapintaa komponentin toimintoihin. Attribuuttien muutoksia voidaan ”kuunnella” *attributeChanged*-funktiossa, jonka parametreina on muuttuneen attribuutin nimi sekä sen vanha ja uusi arvo.

Esimerkki itsenäisestä komponentista on HTML5-standardissa esitelty *video*-elementti [MDN20b]. HTML-sivulle upotettavan mediasoittimen avulla saadaan toistettua mediaa (video/audio) kehittäjän antamien parametrien arvojen (muun muassa median lähde-URL, ruudun koko, automaattinen toisto päällä/pois) mukaan. Toiminnallisuus on

koodattu komponentin sisäiseksi tiedoksi ja kehittäjälle annetaan tietyt attribuutit rajapinnaksi, jolla komponentin toimintoja voidaan muokata: esimerkiksi sitä, toistaako soitin mediaa automaattisesti tiedoston ladattuaan vai vasta käyttäjän klikatessa play-painiketta. Video-elementti on sisäänrakennettu (*built-in*) komponentti eli osa standardia, mutta periaate on sama web-komponenttien kohdalla.

Web Components -teknologiasta erityisen tekee se, että kehittäjä voi koodata komponentteja, jotka ovat osa HTML-standardia ja siitä syystä eivät ole sidottuja mihinkään kolmannen osapuolen kirjastoon. Tämä mahdollistaisi DOMin käytön ohjelmointirajapintana micro frontend -toteutukselle ja sitä ei olisi sidottu mihinkään tiettyyn ohjelmistokehykseen.

Web-komponentit rekisteröidään osaksi DOMin *window*-objektin sisältämää *customElements*-kokoelmaa [MDN20c]. Rekisteröintivaiheessa selaimelle kerrotaan komponentin tägi (*tag*), jolloin selain tietää tuohon nimenomaiseen tägiin liittyvän toteutuksen, suoritettavan koodin. Määrittelyn jälkeen web-komponentti on käytettävissä HTML-koodissa kuin mikä tahansa sisäänrakennettu elementti, esimerkiksi ”video”-, ”div”- tai ”p”-elementti.

Mielenkiintoinen tekijä web-komponenttien spesifikaatiossa on *shadow DOM* -rajapinta [MDN21a]. Tämä teknologia on ollut osa HTML-standardin sisäänrakennettuja komponentteja, kuten edellä mainittua video-elementtiä, jo pitkään. Nyt kyseisen rajapinnan mahdollisuudet on tuotu myös kehittäjien saataville. Sen avulla DOM-puuhun voidaan lisätä haaroja, joiden sisäinen rakenne ja toteutus, mukaan lukien tyylimäärittelyt, ovat ”piilossa” ulkopuolelta. Se tarkoittaa, etteivät esimerkiksi css-valitsimet löydä elementtiä shadow DOMin ulkopuolelta. Tämä tarjoaa uudenlaiset mahdollisuudet jakaa koodia itsenäisiin, eristettyihin osiin myös DOM-puussa. Shadow DOMin käyttö ei kuitenkaan ole pakollista. Jos esimerkiksi halutaan komponenttien noudattavan globaaleja tyylimäärittelyjä, voidaan komponentit koodata osaksi normaalia DOMia.

Riippumatta shadow DOMin käytöstä tarjoavat web-komponentit vaihtoehdon esimerkiksi iframe-elementeille [MDN20a]. Iframe on suunniteltu mahdollistamaan kokonaisen HTML-dokumentin upottamisen toiselle HTML-sivulle. Upotettu HTML-sivu muodostaa oman selailukontekstinsa (*browsing context*), mikä tarkoittaa muun muassa sitä, että selain allokoi jokaiselle iframe-elementille muistia ja CPUn laskentaa

erikseen. Jokaiselle iframe-elementille syntyy muun muassa oma Document API ja sen myötä DOM-rakenne. Siksi näiden hallinnointi ohjelmallisesti voi tulla haastavaksi, ja sivusto voi kohdata suorituskykyongelmia, vaikka itse iframessa näytettävä sisältö olisikin vähäistä. Web-komponentit taas on suunniteltu siten, että kokonaisen HTML-sivun sijaan elementti koostuu vain tarvittavasta HTML-osiosta, esimerkiksi ”div”-elementistä lapsielementteineen.

Tämän teknologian testaaminen oli ensimmäisten demototeutusten aiheena. Koska aiempi kokemus täysin komponenttipohjaisesta toteutuksesta oli Reactista, oli demovaiheessa mielenkiinnon kohteena erityisesti koestaa, kuinka web-komponentit vertautuisivat näihin; voisiko web-komponenteista rakentaa vastaavia toiminnallisia komponentteja.

Mikäli edellä mainittujen lisäksi niiden onnistuisi suorittaa ajax-kutsu[10] palvelimelle ja tulostaa hakemansa data, niin tällöin lähes kaikki tarvittavat elementit itsenäisten komponenttien rakentamiseen olisi kasassa. Ensimmäiset demot osoittivat teknologian noilta osin toimivaksi.

Web-komponentit määritellään käyttäen ES6:n [ECM15] luokkasyntaksia (Kuva 3.2). JavaScriptin – niin ikään ES6-spesifikaatiossa esitellyn – moduulisyntaksin myötä yksittäisiä tiedostoja voidaan ladata käyttöön toisissa moduuleissa. Niinpä omat web-komponentit voidaan ladata joko script-tägilla tai import-syntaksilla omasta URL-soitteestaan ja käyttää niitä tarpeen mukaan.

[10] Asynchronous JavaScript and XML eli lyhyemmin Ajax on kokoelma web-teknologioita, joiden avulla web-sovellus voi kommunikoida, eli hakea ja lähettää tietoa pienissä erissä, palvelimen kanssa ilman koko sivuston uudelleen lataamista. Lähde: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

```
1 <body>
2     <!-- komponentin käyttö DOMissa -->
3     <esim-komponentti></esim-komponentti>
4
5     <script>
6             // komponentin määrittely tapahtuu
7             // ECMAScript2015:n (ES6) luokkasyntaksia käyttäen
8             class EsimKomponentti extends HTMLElement {
9
10                constructor() {
11                    super();
12
13                    // shadow rootin määrittely
14                    const shadow = this.attachShadow({ mode: 'open' })
15
16                    // sisältöä
17                    const container = document.createElement('div');
18                    container.innerHTML = "<h1>Hello world!</h1>"
19
20                    shadow.appendChild(container);
21                }
22            }
23            // komponentin rekisteröinti
24            customElements.define('esim-komponentti', EsimKomponentti);
25    </script>
26 </body>
```

Kuva 3.2. Yksinkertaisen web-komponentin määrittely luokkasyntaksia käyttäen (rivi 8), rekisteröinti (rivi 24) sekä käyttö document object modelissa, DOMissa (rivi 3). Tämä komponentti tulostaa HTML-sivulle otsikkotason (h1) tekstin ”Hello world!”.

Staattiselle *import*-skriptille on tulossa[11] dynaaminen vastine, *dynamic import*, jolla JavaScript-moduuleita voidaan ladata ohjelman suorituksen aikana, dynaamisesti. Olipa lataus sitten staattinen tai dynaaminen, se kuitenkin tarkoittaa sitä, että HTML-sivulle voidaan ladata web-komponentti ulkopuolisesta *url*-osoitteesta ja käyttää sitä kuten muitakin HTML-elementtejä.

Itsenäiset web-komponentit yhdistettynä moduuliteknologiaan muodostavat tämän tutkielman aiheen, micro frontend -arkkitehtuurin implementaation pohjateknologian. Tässä vaiheessa esimerkiksi iframe-elementtien käytön saattoi hyvällä syyllä karsia kandidaattien joukosta pois. Web-komponenttien soveltuvuus käyttöliittymätason integraatioon on omaa luokkaansa jo pelkästään suorituskykynsä (muistinvaraus) puolesta. Oli myös tiedossa, että iframe-elementtien sisällön skaalaus näytön koon

[11] *Dynamic import*: https://tc39.es/proposal-dynamic-import/

mukaan saattaisi jatkossa aiheuttaa ongelmia etenkin mobiililaitteiden kohdalla.

Mainittakoon, että Zalandon blogissa [Zal18] web-komponentit yhdistettiin vielä palvelinpäässä tapahtuvaan renderöintiin *Server Side Includes* -skriptillä [SSI20]. Asukassivujen kohdalla tarkoitus oli välttää palvelinpään renderöintiä ja pyrkiä kommunikoimaan palvelimen kanssa vain REST-rajapinnan välityksellä.

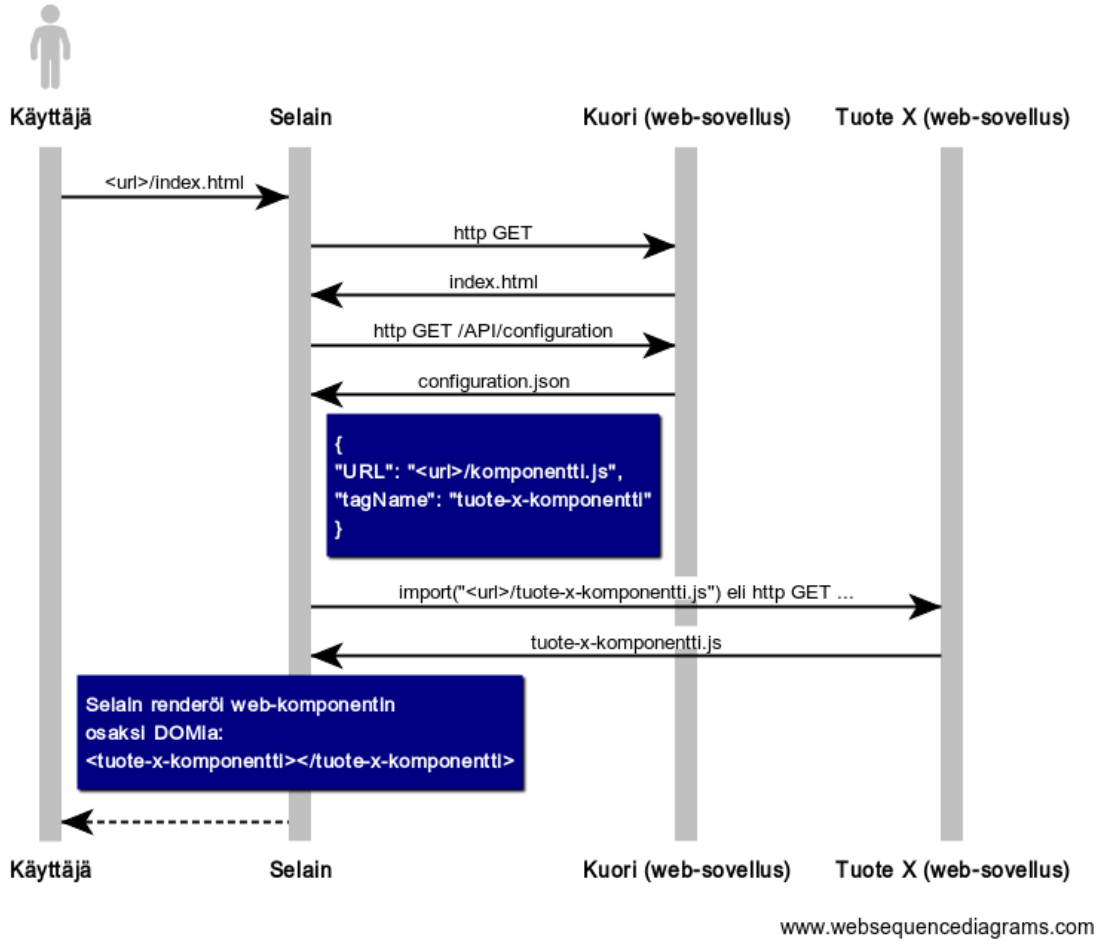
Konfiguraatiopalvelu tulisi olemaan tärkeässä roolissa kahdestakin syystä. Ensiksi, uusi toteutus tulee toimimaan moniasiakasmallisesti eli yksi instanssi palvelee montaa asiakasta. Tästä johtuen esimerkiksi näytettävien mikropalveluiden kokoonpano ja sovelluksen ulkoasu vaihtelevat loppukäyttäjän asiakastilin mukaan. Toiseksi, arkkitehtuurin on tuettava helpompaa käyttöönottoprosessia ja ylläpitoa: käyttöönotto voidaan tehdä ilman tuotantokatkoja ja tarvittavat asetukset tehdään juuri konfiguraatiopalvelun kautta Asukassivujen ollessa käynnissä.

Ensimmäiset demototeutukset antoivat suuntaa web-komponenttien soveltuvuudesta mikropalveluiden käyttöliittymien rakennuspalikoiksi erityisesti näiden elinkaarifunktioiden osalta. Seuraavana oli komponenttien lataus ja käyttö ulkoisesta lähteestä, millä simuloitiin jo tulevan micro frontend -arkkitehtuurin *application shell* -suunnittelumallia[12]. Usein *progressive web application* -tyyppisen (PWA) ohjelmiston yhteydessä mainittu application shell -arkkitehtuurimalli tarkoittaa ”sovelluskuoren” ja varsinaisen sisällön erottamista toisistaan. Päämääränä tässä on muun muassa saada ladattua sovelluksen käyttöliittymä nopeasti ilman, että käyttöliittymä ”jäätyy” mahdollisen sisältödatan hakemisen ajaksi. Tuolloin, kesän ja alkukevään 2018 aikana, application shell -arkkitehtuurimallia ei oltu mainittu micro frontendiä koskevissa artikkeleissa. Ensimmäinen maininta on micro-frontends.org-sivuston ylläpitäjän, Michael Geersin, toistaiseksi julkaisemattomassa, Micro Frontends in Action -kirjassa [Gee19].

Ensimmäisissä POC-toteutuksissa (Kuva 3.3) riitti simuloida konfiguraatiopalvelua yksinkertaisella REST-kutsulla. Palvelu A:lta ladataan selaimelle HTML-sivu. Sivulta suoritettava ajax-kutsu hakee REST-rajapinnasta konfiguraation, jossa on sivulle ladattavan web-komponentin osoite (palvelussa B) ja tägin nimi. Selaimessa suoritettava koodi hakee web-komponentin (aiemmin mainittu *import*) ja renderöi sen annetulla tägin

[12] Progressive web app, application shell -arkkitehtuuri, https://developers.google.com/web/updates/2015/11/app-shell

nimellä DOMiinsa.



Kuva 3.3. Sekvenssikaavio yksinkertaisesta proof of concept –toteutuksesta. ”Kuori” ja ”Tuote X” ovat itsenäisiä web-sovelluksia. Kuori toimii palveluna, joka toimittaa sekä staattisen html:n että JSON-muotoisen konfiguraation. Tuote X toimittaa web-komponentein toteutetun näkymän JavaScript-modulina. Kuori toimii siis molempien sovellusten asiakaspäänä eli käyttöliittymätason integraatiokerroksena. JSON-muotoinen data simuloi konfiguraatiota, joka lopullisessa toteutuksessa haettaisiin konfiguraatiopalvelusta.

Suunnittelupäätös 4 (SP4): LitElement

Tavanomaisen SPA-toteutuksen voisi tehdä millä tahansa suhteellisen tunnetulla JavaScript-kehyksellä ilman, että tarvitsee tehdä kovinkaan tarkkoja vertailuja näiden välillä. Valitsemalla esimerkiksi Reactin, Vuen tai Angularin saa varmuudella kehyksen, joka vastaa suurimpaan osaan web-kehityksessä eteen tulevista haasteista. Valintaa voisi painottaa vaikkapa sovelluskehittäjien aiemmalla kokemuksella.

Web-komponenteista tietoa hakiessa törmäsi varsin nopeasti Polymer-nimiseen JavaScript-kirjastoon. Googlen Polymer-projektin [PoP20] kehittämä avoimen lähdekoodin kirjasto pohjautuu komponentteihin, mutta toisin kuin esimerkiksi Reactissa,

sen komponentit ovat HTML-standardin mukaisia web-komponentteja. Vaikka edellä kuvatut POC-toteutukset ajoivat suunnittelua vahvasti web-komponenttien suuntaan, ei mikään varsinaisesti vielä sulkenut pois esimerkiksi Reactin käyttöä. Se on erittäin laajasti käytetty JavaScript-kirjasto, joka soveltuu erinomaisesti modernien web-sovellusten toteuttamiseen.

HTML-standardin mukaisten web-komponenttien ja React-komponenttien merkittävimmät erot liittyvät niiden DOMin käsittelyyn ja renderöintiin. React hallinnoi DOMia pitämällä muistissa sen kopiota, virtuaalista DOMia. Kehittäjällä on näin pääsy ”oikeaan” DOMiin vain tämän kautta. Web-komponentit sen sijaan laajentavat suoraan itse HTML-alustaa, jolloin myös DOMin päivitys tehdään suoraan niillä.

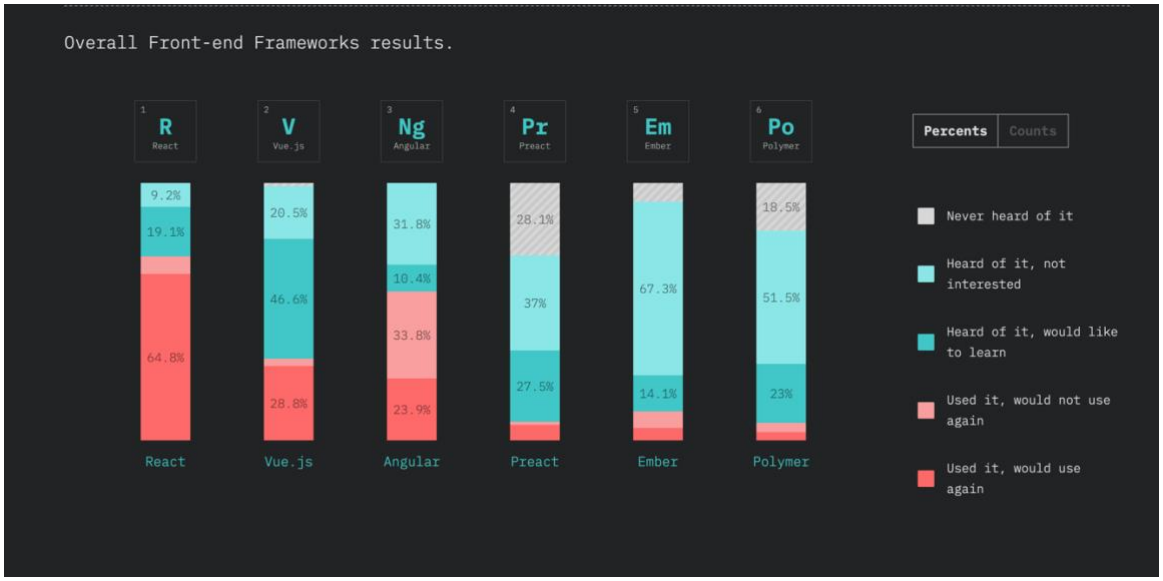
React-komponentit renderöidään useimmiten käyttäen JSX-laajennetta (JavaScript Xml), jonka avulla React-sovelluksissa kirjoitetaan HTML-koodia. Web-komponentit taasen sallivat minkä tahansa kirjaston tai kehyksen käytön, joka tukee standardia. Reactin pitäisi pystyä toimimaan myös web-komponenttien kanssa, mutta toteutus ei ole tällöin aivan suoraviivaista [REA20b].

Polymerin filosofia on ollut nojata standardiin (Web Components API) ja korvata polyfilleillä[13] eri selainvalmistajien jättämät aukot teknologian tukemisessa [PoL20]. Sitä mukaa kun HTML-standardiin hyväksytyt ominaisuudet tulevat tuetuiksi käytetyimpien selainten JavaScript-moottoreissa, tarve polyfilleille vähenee. Niinpä kirjasto pyrkii pienentämään kokoaan näiltä osin ajan mukaan.

Polymer-kirjasto oli huomattavasti tuntemattomampi kuin React (Kuva 3.4), vaikka silläkin on suuri yritys (Google) takanaan ja sitä on käytetty nimekkäissä palveluissa (YouTube, Google Earth, Netflix, General Electric)[14]. Polymeristä oli julkaistu ensimmäinen versio keväällä 2015 ja kesän 2018 aikana siitä oli julkaistu jo toinen niin sanottu *major*-versio 2.0 ja kolmaskin, 3.0, oli jo työn alla.

[13] Polyfillillä tarkoitetaan vaihtoehtoista toteutusta toiminnallisuudelle, jota ei jonkun selaimen JavaScript-moottori tue

[14] Kattava listaus Polymeriä käyttävistä yrityksistä ja sovelluksista: https://github.com/Polymer/polymer/wiki/Who's-using-Polymer%3F



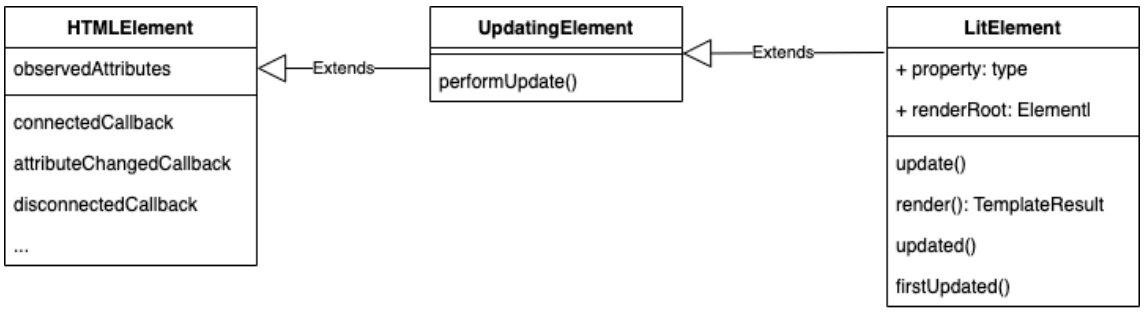
Kuva 3.4. Suuntaa antava kuva frontend-kehysten tunnettavuudesta. Kuvakaappaus kyselystä ”State of JS” vuoden 2018 tuloksista. React oli tuolloin tunnetuin frontend-ohjelmistokehys: ”Used it, would use again”: lähes 65% vastaajista, Polymerillä vastaava luku 3.1%. Polymerillä myös ”Never heard of it”: 18.5% ja vastaava Reactilla 0%. Lähde: https://2018.stateofjs.com/front-end-frameworks/overview/

Web-komponentit on esitetty yhtenä mahdollisena toteutusmallina micro frontend -arkkitehtuurille muutamissa artikkeleissa [FoL14, Gee19, PAS20]. Vaihtoehtoiset toteutustavat on tehdä ne joko ohjelmoimalla itse (*pure JavaScript*) tai ohjelmistokehyksien avulla. Toistaiseksi web-komponentit ovat jääneet vain maininnoiksi, ja varsinaiset prototyyppitoteutukset julkaistujen artikkelien taustalla ovat perustuneet ohjelmistokehyksiin, muun muassa Reactiin [PAS20] ja Angulariin [HRI17].

Voisi väittää, että etenkin SPA-ratkaisua tehdessä olisi jopa kyseenalaista jättää hyödyntämättä jonkin ohjelmistokehyksen hyödyt ja koodata kaikki tarpeellinen itse. Web-komponentteihin perustuvassa ratkaisussa on kuitenkin hyvä puntaroida sitä seikkaa, tuoko erillinen kehys merkittävää hyötyä yksittäisten komponenttien toteutuksessa ja niiden käyttämisessä. Olisi perusteltua argumentoida jopa tätä vastaan; sitä mukaa kun jonkun kirjaston piirteet tulevat osaksi standardia, perustelut kyseisen kirjaston käytölle vähenevät. HTML:n kontekstissa tämä on varsin tavallista. Esimerkiksi jQuery alkaa olla suhteellisen tarpeeton, koska JavaScript tarjoaa käytännössä vastaavat css-valitsimet. Myös tarve erilaisille apukirjastoille[15] on vähentynyt JavaScriptin kehittyessä.

[15] Esimerkiksi kirjastot Lodash ja Underscore tarjoavat paljon funktioita kokoelmien käsittelyyn. Nykyään vastaavia löytyy jo suoraan JavaScriptistä. Lähteet: http://lodash.com, http://underscore.org.

Polymer-kirjastolle oli paljon puoltavia argumentteja, varsinkin kun web-komponentit oli jo todettu kelvolliseksi pohjateknologiaksi käyttöliittymäkerroksen integraatiossa. Polymer-projektin Gray Nortonin blogikirjoitus [Nor18] kirjaston tulevaisuudesta oli kuitenkin ratkaiseva teknologiavalinnan suhteen. Blogissa suositeltiin vanhojen Polymer-pohjaisten projektien kohdalla migraatioita uuteen 3.0-versioon, mutta uusien kohdalla suositeltiin uuden *LitElementin* [LIT20a] käyttöä. Kuitenkin sillä varauksella, että olisi valmis käyttämään ennakkoversiota tästä – vahvasti kehitysvaiheessa (tuolloin versiossa 0.5.2) – olevasta kantaluokasta web-komponentille (Kuva 3.5). Tässä vaiheessa oli siis tehtävä valinta kahden suhteellisen kehitystiimille tuntemattoman teknologian välillä: Polymer 3.0 vai LitElement, joista jälkimmäinen oli käytännössä vieläkin tuntemattomampi.



Kuva 3.5. LitElement on kantaluokka web-komponentille. HTMLElement on HTML-standardissa määritelty rajapinta, jota laajentamalla kehittäjä voi tehdä omia web-komponentteja. LitElement muun muassa tuottaa automaattisen päivityssyklin komponentin kenttien muuttaessa arvoaan.

LitElementin tuotantojulkaisulle ei oltu annettu tarkkaa päivämäärää, mutta riski esimerkiksi koko teknologian takaisinvetämiselle nähtiin hyvin pienenä; taustalla oli Googlen Polymer-projekti, joka oli jo vuosia ollut web-komponenttiteknologian suurimpia puolestapuhujia. Lisäksi, Googlen omat ohjelmistot - muun muassa You Tube - käyttivät web-komponentteja. Se että tuotantoversiota ei tulisikaan, olisi joka tapauksessa pieni riski; LitElementin ainoa riippuvuus on saman tiimin julkaisema *lit-html*-kirjasto [LIT20b], jonka avulla komponentin sisältö renderöidään DOMiin. Muita riippuvuuksia ei ollut, joten sellaisten vanhenemisesta ei tulisi riskiä, vaikka kehitys loppuisi jo ennen ensimmäisen tuotantoversion julkaisua.

LitElement valikoitui apukirjastoksi web-komponentteihin pohjautuvassa toteutuksessa. Sen käyttöönoton riskit olivat suhteellisen pienet verrattuna sen tuomaan hyötyyn. Esimerkkinä vaikkapa reagointi komponentin tilanmuutoksiin: natiivien web-

komponenttien kohdalla renderöinti tapahtuu elementin innerHTML- tai textContent-kenttää manipuloimalla itse koodatun logiikan mukaisesti. LitElementiin toteutettu *observer*-suunnittelumalli[16] mahdollistaa komponentin kenttien tilamuutosten tarkkailun ja automaattiset päivityssyklit. Jonkin kentän arvon muuttuminen käynnistää automaattisesti muun muassa *render*-funktion suorituksen. Monimutkaisemmissa komponenteissa jo pelkästään tämä piirre vähentää tarvetta ohjelmakoodin kirjoittamiselle. Toisaalta, jos jokin komponentti ei tarvitse juurikaan logiikkaa, voi sen kirjoittaa edelleen natiivina web-komponenttina.

Suuri hyöty tulee siitä, että sovelluskuoren ja jokaisen siihen ladattavan palvelun välillä ei tule olemaan vahvaa HTML-standardin ulkopuolista teknologista sidosta. Ensimmäisellä näistä on merkitystä erityisesti ohjelmiston elinkaaren kannalta: Asukassivujen ylläpitotyöt eivät ole riippuvaisia kolmannen osapuolen kirjaston julkaisusykleistä. Toisaalta tämä ei myöskään aseta rajoitteita sille, etteikö mikropalvelun kehityksessä voisi käyttää jotain kehystä: jos esimerkiksi yritys haluaisi ostaa valmiin, Reactilla toteutetun varausjärjestelmän, integraatioon riittäisi sen ”kääriminen” web-komponentiksi. Koko Web Components -teknologian perusperiaatteisiin kuuluu toteutuslogiikan kapselointi, ja niin kauan kuin käytettävät apukirjastot tai ohjelmistokehykset eivät ole ristiriidassa HTML-standardien kanssa, kaiken pitäisi toimia.

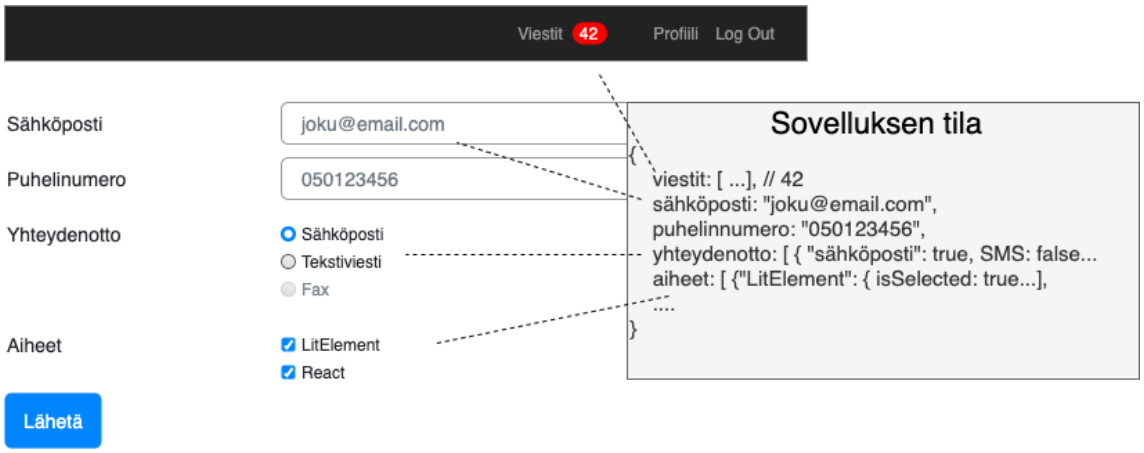
Rajoitteita toki voi tulla muista syistä. Jos esimerkiksi jokin micro frontend -tuote käyttäisi Reactia, se tarkoittaisi sitä, että koko React-kirjasto ja sen mukanaan tuomat riippuvuudet olisivat osa ”verkon yli” ladattavaa koodia. Eli kaikissa tilanteissa tällainen ei olisi järkevää, vaikka se olisikin mahdollista.

Suunnittelupäätös 5 (SP5): keskitetty tilanhallinta - Redux.js

Mitä monimutkaisemmiksi komponentit ja niistä koostuvat sovellukset kasvavat, sitä hankalammaksi tulee myös niiden tilanhallinta. Tilalla tarkoitetaan sovelluksen käyttöliittymäkomponenttien arvoja (Kuva 3.6). Nämä arvot muuttuvat käyttäjän

[16] Observer-suunnittelumalli on toteutettu useisiin ohjelmistokirjastoihin ja -kehyksiin. Usein termi ”observable” näissä yhteyksissä kuvaa juuri tätä ominaisuutta. Lähteitä: https://en.wikipedia.org/wiki/Observer_pattern Esim. Knockout.js: https://knockoutjs.com/documentation/observables.html Angular: https://angular.io/guide/observables

toiminnan seurauksena.



Kuva 3.6. Kuvitteellinen esimerkki sovelluksen tilasta. Tilalla tarkoitetaan yhden tai useamman käyttöliittymäkomponentin sen hetkistä dataa. Esimerkissä yläpalkin viestien lukumäärä ja näkyvän lomakkeen arvot yhdessä muodostavat sovelluksen tilan. Käyttäjän toiminta muuttaa tilaa, ja jonkun käyttöliittymäkomponentin tila voi riippua toisen komponentin tilasta. Esimerkiksi ”Lähetä”-painike voisi olla disabloitu, mikäli Sähköposti-kentän ja Yhteydenotto-valinnan välillä olisi ristiriita.

Yksinkertaisemmissa tilanteissa lomakkeiden arvot voivat tallentua sellaisenaan tietokantaan, ja sovelluksen tila on yhtä kuin viimeksi haettu tieto. Asukassivujen kohdalla, jossa tieto on hajautunut useaan palveluun ja jossa jopa näytettävien tuotteiden kokoonpano vaihtelee, tilanhallinnalle tulee enemmän vaatimuksia.

Redux [RED20] on kirjasto, joka on kehitetty sovelluksen tilanhallintaan. Kokemusta vaihtoehtoisista kirjastoista ei juuri ollut. React-kehityksen myötä oli käynyt selväksi, että komponenttien määrän kasvaessa on järkevää pyrkiä keskitettyyn ratkaisuun, ”yhteen totuuden lähteeseen”[17]. Etenkin, jos sovelluksen tilan tietoja tarvitaan useissa paikoissa. Tilanne oli nähtävissä näinkin hajautetussa ratkaisussa.

Reduxin keskeiset elementit ovat *actionit*, tila, jota edustaa sovelluksen käsittelemät tiedot ja näkymät. *Action* tarkoittaa tapahtumaa, jolla sovelluksen tilaa muutetaan. Näkymät ”kuuntelevat” tilan muutoksia ja päivittyvät uuteen tilaan. Reduxin *middlewaret* mahdollistavat tiettyjen toistuvien toimintojen siirtämisen taustalle: esimerkiksi käyttäjän istunnon voimassaolon tutkimisen tai sivustolla navigoinnin ja sen perusteella tapahtuvan lokituksen.

Sovelluksen tilanhallinnan keskittäminen Reduxille edesauttaa osaltaan Asukassivujen kehittämistä ilman web-ohjelmistokehystä. Käyttöliittymän osalta voitaisiin puhua jopa

[17] Yksi Reduxin kolmesta periaatteesta: ”single source of truth”: https://redux.js.org/understanding/thinking-in-redux/three-principles

”kehyksettömästä” (*frameworkless*) ratkaisusta.

Yhteenveto

Edellä kuvattiin suunnitteluprosessin etenemistä kronologisesti teknologiaselvityksistä prototyyppitoteutuksiin ja teknologiavalintoihin. Esille nostettiin tärkeimmät linjaukset ja taustoitettiin perusteluja tärkeimpien suunnittelupäätösten taustalla (Taulukko 3.2).

Moniasiakasmalliin siirtyminen vähentäisi sovellusinstanssien määrää ja konfiguraatiopalvelun vastuulle tulisi säilyttää asiakaskohtaiset asetukset (L1 ja L2). Nämä olivat ajureita konkreettisemmille suunnittelupäätöksille (SP1-SP5), joiden tuli edistää linjausten mukaista lopputulosta. Kokonaisarkkitehtuuriin kannalta tärkeässä asemassa on myös liitännäispalveluiden muodostama mikropalvelukerros (MP). Yleistetään myös itse micro frontend -osuus omaksi suunnittelupäätöksekseen (MF), jolloin se voidaan nähdä selkeästi osana kokonaisarkkitehtuuria, ja josta se korvaa Umbraco-pohjaisen arkkitehtuurin.

Linjaus (L) / Suunnittelupäätös (SP)	**Selitys**
L1	Moniasiakasmalliin siirtyminen.
L2	Konfiguraatiopalvelun roolin kasvattaminen
SP1	SPA, *single-page application*
SP2	CSR, *client-side rendering*, asiakaspään renderöinti (vs. SSR)
SP3	Komponenttipohjainen arkkitehtuuri web-komponentein
SP4	Ei frontend-ohjelmistokehystä, teknologiavalinta: LitElement
SP5	Tilanhallinta: Redux.js
MP	Mikropalveluarkkitehtuuri ja liitännäispalvelut
MF	Micro frontend -arkkitehtuuri

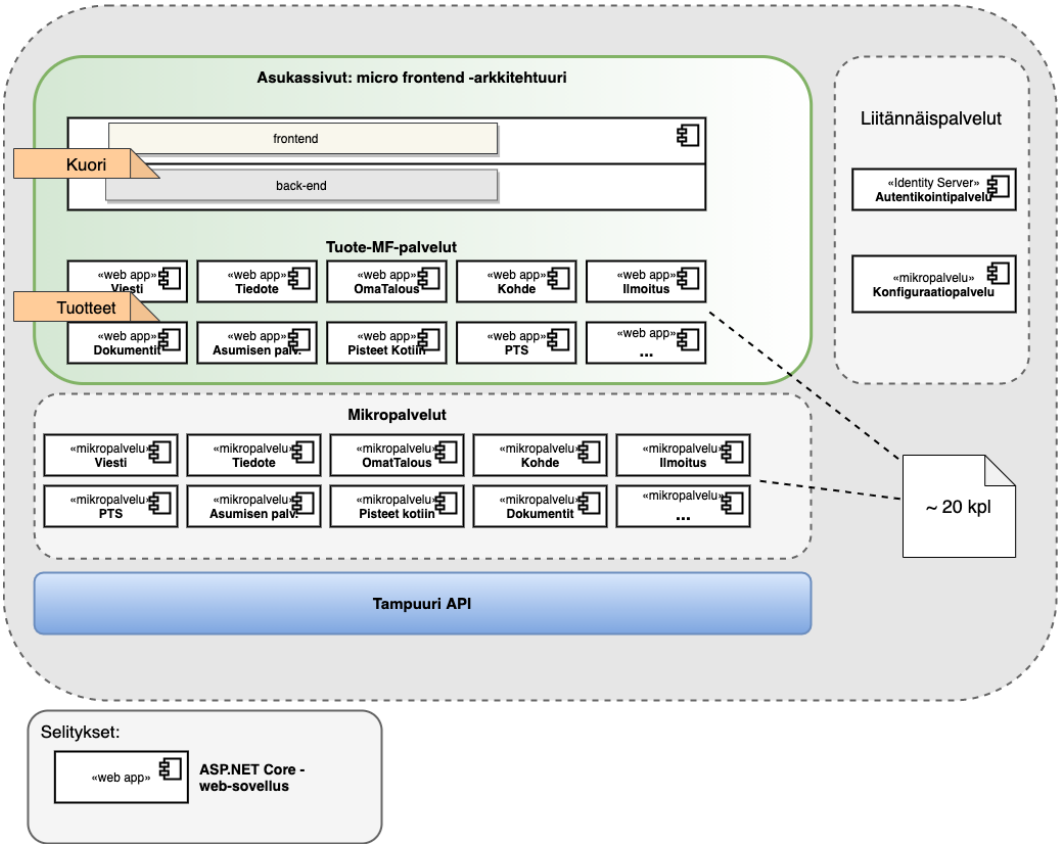
Taulukko 3.2. Tärkeimmät linjaukset ja suunnittelupäätökset.

3.2 *Implementaatio*

Tässä luvussa kuvataan Asukassivujen kokonaisarkkitehtuuri ja erityisesti sen micro frontend -osuus eli tutkielman varsinainen aihe. Aluksi käydään läpi Asukassivujen kokonaisarkkitehtuurin komponentit ja niiden välinen kommunikaatio. Sen jälkeen otetaan micro frontend -osuus lähempään tarkasteluun. Käyttöliittymä ja sen muokattavuus ovat keskeisessä roolissa, joten näitä tarkastellaan useammasta näkökulmasta. Lopuksi vedetään kompleksinen kokonaisuus yhteen peilaten sitä aiemmin esitettyyn prototyyppimalliin.

Asukassivujen kokonaisarkkitehtuuri: komponentit ja vastuut

Asukassivut-ohjelmisto tarvitsee toimiakseen Tampuuri APIn datan, mikropalvelut sekä liitännäispalvelut autentikointia ja konfigurointia varten. Näistä muodostuu Asukassivujen kokonaisarkkitehtuuri (Kuva 3.7 ja Taulukko 3.3). Micro frontend -arkkitehtuurimalli korvaa aiemman monoliittisen Umbraco-pohjaisen arkkitehtuurin, joka esiteltiin aliluvussa 2.2.



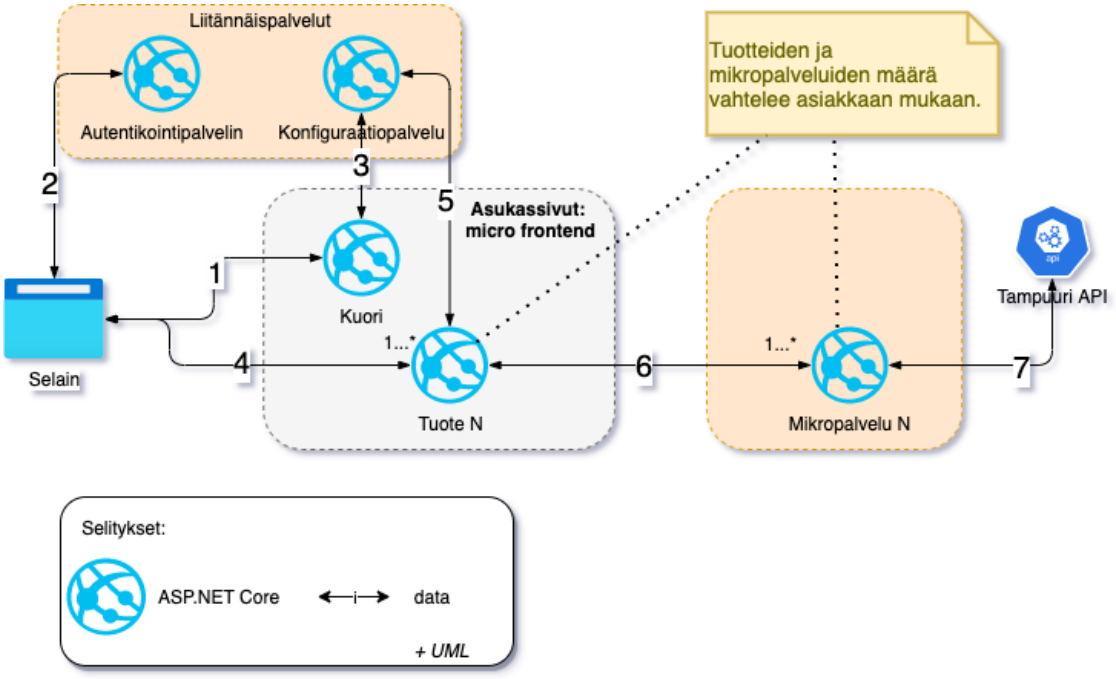
Kuva 3.7. Asukassivujen kokonaisarkkitehtuuri. Asukassivut, mikropalvelut, liitännäispalvelut sekä Tampuuri API. Asukassivut koostavat palvelut on kuvattu suurpiirteisesti: Kuori sekä osajoukko Tuote-micro frontend -palveluista (4/2021 noin 20 eri palvelua). Taulukko 3.3 kuvaa komponenttien vastuut.

Komponentti	Päävastuut
Kuori	1.) Toimii asiakaspäänä (asiakas-palvelin) kaikille tuotteille (sivustorakenne, navigoituminen, sivun layout), 2.) pitää yllä sovelluksen tilaa (Redux.js), 3.) valvoo käyttäjän kirjautumista, 4.) orkestroi tuote-MF-palveluiden käyttöä.
Tuotteet (Kuvassa 3.7 ”Tuote-MF-palvelut”)	Tuote 1.) tarjoaa käyttöliittymäkomponenttien koodit omina tiedostoinaan, 2.) toimii APIna tuotteen datalle, 3.) toimii välikerroksena Kuoren asiakaspään sekä Mikropalvelun välillä.
Mikropalvelu	1.) Välittää dataa (rest/json) Tampuuri APIn ja samaa liiketoiminta-aluetta edustavan tuotepalvelun välillä.
Tampuuri API	Tampuuri-toiminnanohjausjärjestelmän rajapintapalvelut.
Autentikointipalvelu	Käyttäjän rekisteröinnit ja kirjautumiset.
Konfiguraatiopalvelu	Toimittaa suoritusaikana kaikkien palveluiden asiakaskohtaiset asetukset (konfiguroinnit).

Taulukko 3.3. Kokonaisarkkitehtuurin (Kuva 3.7) komponentit ja niiden vastuut.

Kommunikaatio kokonaisarkkitehtuurissa

Kuvan 3.8 komponenttien kommunikaatio selitetään sitä seuraavassa Taulukossa 3.4. Datavirtojen tunnukset kuvassa vastaavat taulukon ensimmäisen sarakkeen numeroita.



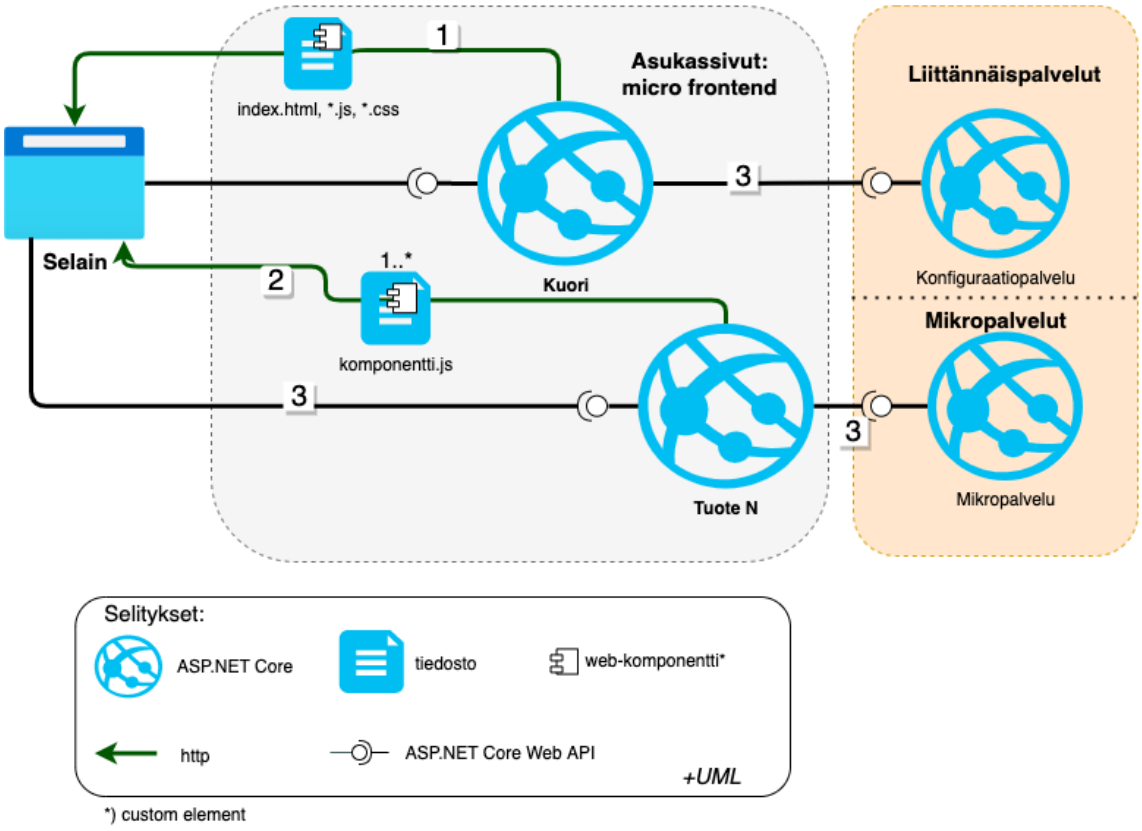
Kuva 3.8. Komponentit ja kommunikaatio. Taulukossa 3.4 kuvataan datavirrat tarkemmin.

#Tunniste	Selitys
1	Kuori käsittelee selaimelta tulevat http-pyynnöt.
2	Käyttäjän autentikointi suoritetaan Autentikointipalvelimella.
3	Kuori lataa autentikoituneen käyttäjän asiakastilin mukaisen konfiguraation ja muodostaa sivuston (#1 selaimelle).
4	Tuote N toimittaa käyttöliittymän (”bundlattu” JavaScript-tiedosto) ja datan (JSON).
5	Tuote N hakee sitä koskevat asiakaskohtaiset asetukset (#4 selaimelle).
6	Tuote N kommunikoi sen vastinparina toimivan Mikropalvelun kanssa.
7	Mikropalvelu toimii välikerroksena Tampuuri APIn ja Tuote-palvelun välillä. Se muun muassa mallintaa Tampuuri APIn tuottaman datan Asukassivujen tarvitsemaan muotoon.

Taulukko 3.4. Kommunikaatio kokonaisarkkitehtuurissa.

Kommunikaatio micro frontend -arkkitehtuurissa

ASP.NET Core -kehys tukee REST-rajapintojen toteutusta sekä myös SPA-toteutusmallia, jolloin ne voivat palvella staattisia tiedostoja käyttöliittymälle. Työnjako menee näiden osalta siten, että vain Kuorelta tulee HTML-tiedosto (sekä omat muut asiakaspään koodinsa) ja jokainen Tuote ”hostaa” oman käyttöliittymänsä staattisina JavaScript-tiedostoina. Kuvan 3.9 kommunikaatio avataan tarkemmin Taulukossa 3.5. Kuvan ja taulukon yhteys samoin kuin edellisessä kohdassa.



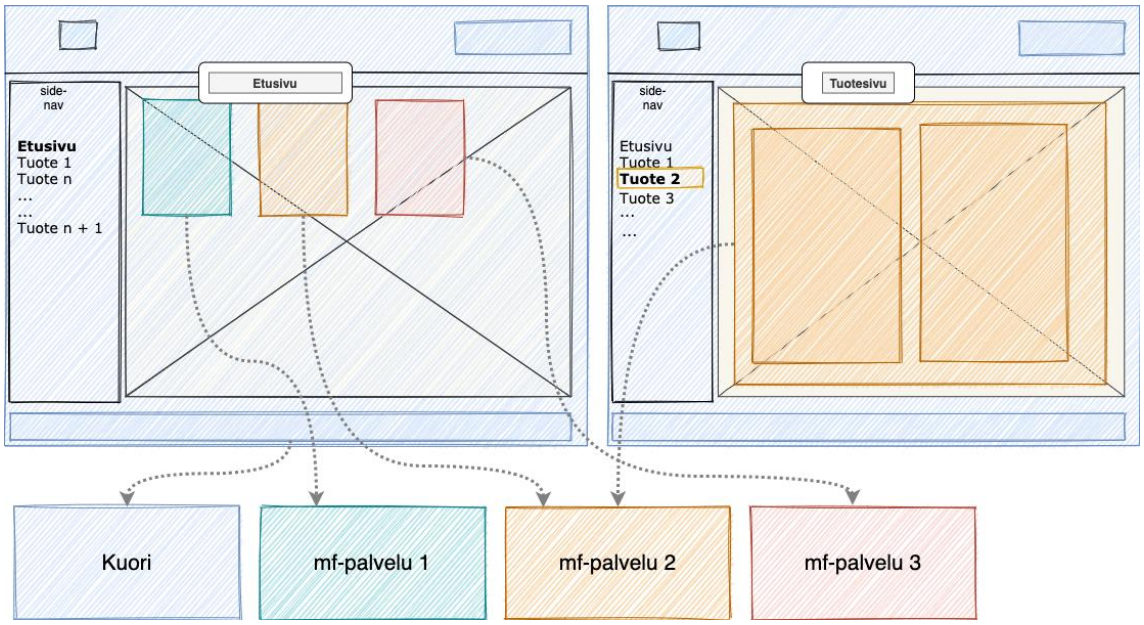
Kuva 3.9. Kommunikaatio Asukassivujen eli micro frontend -arkkitehtuurin tasolla. Taulukossa 3.5 kuvataan kommunikaatio tarkemmin.

# **Tunniste**	**Selitys**
1	Kuori toimittaa html-sivun, tyylit (css-tiedoston) sekä ”bundlatut” JavaScript-tiedostot selaimelle.
2	Tuote N toimittaa web-komponentit ”bundlattuina” JavaScript-tiedostoina.
3	Jokainen web-palvelu (myös Tuote) toimittaa REST-rajapinnan (ASP.NET Coren web API) dataa varten.

Taulukko 3.5. Kommunikaatio selaimen ja Asukassivujen keskeisten palveluiden välillä.

Käyttöliittymä ja palvelut

Micro frontend -arkkitehtuurissa palvelut integroidaan yhdeksi kokonaisuudeksi käyttöliittymätasolla (Kuva 3.10). Kuori toimittaa käyttöliittymän peruskehikon: muun muassa otsikkopalkin, alapalkin ja navigaatioelementin. Varsinainen sisältö muodostuu Kuoreen ladattavien tuotteiden mukaan. Tuotetta edustaa oma micro frontend -palvelunsa (kuvassa ”mf-palvelu”). Tuotekokoonpano riippuu asiakaskohtaisesta konfiguraatiosta. Sivuston navigaatio muodostetaan dynaamisesti tuotekokoonpanon mukaisesti.

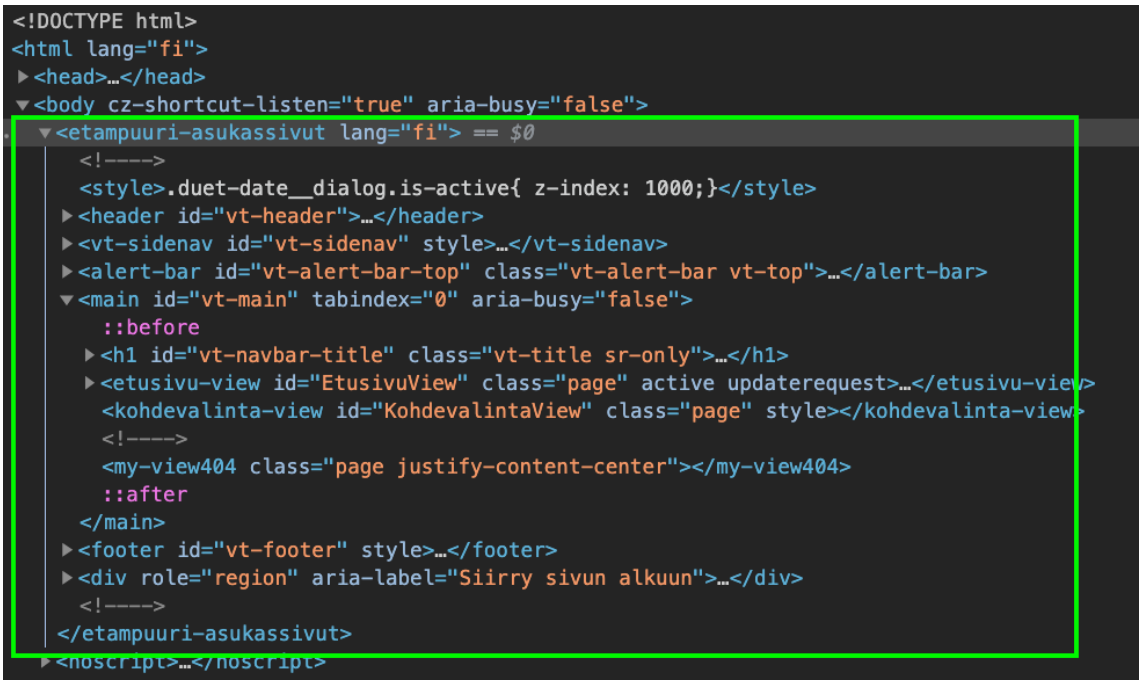


Kuva 3.10. Asukassivujen käyttöliittymän kuvaus suhteessa micro frontend -palveluihin (”mf-palvelu”). Vasemmanpuolinen näkymä esittää etusivua, jossa näytetään kooste useamman tuotepalvelun tilasta pienillä yhteenvetokomponenteilla, ”tiilillä”. Oikeanpuoleinen kuva edustaa tuotenäkymää, jolloin käyttöliittymälle ladataan varsinainen Tuote-sovellus eli komponentti, joka koostuu komponenteista.

Web-komponentit käyttöliittymässä

Kuori

Nimensä *application shell* -arkkitehtuurimallilta saanut web-palvelu on myös SPA-sovellus. Sen päätasolla (rungossa, HTML:n *body*) on web-komponentti (*etampuuri-asukassivut*), joka koostuu monista muista sisäisistä komponenteista (Kuva 3.11). Se muodostaa sivun *layoutin* ja luo sisällön konfiguraatiosta saatujen ”ohjeiden” mukaisesti. Tämä päätason komponentti on kytketty Redux-tilanhallintaan, jonka välikerroksien (*middlewaret*) vastuille on delegoitu muun muassa käyttäjän autentikoinnin valvonta ja monitorointiin liittyviä toimintoja.

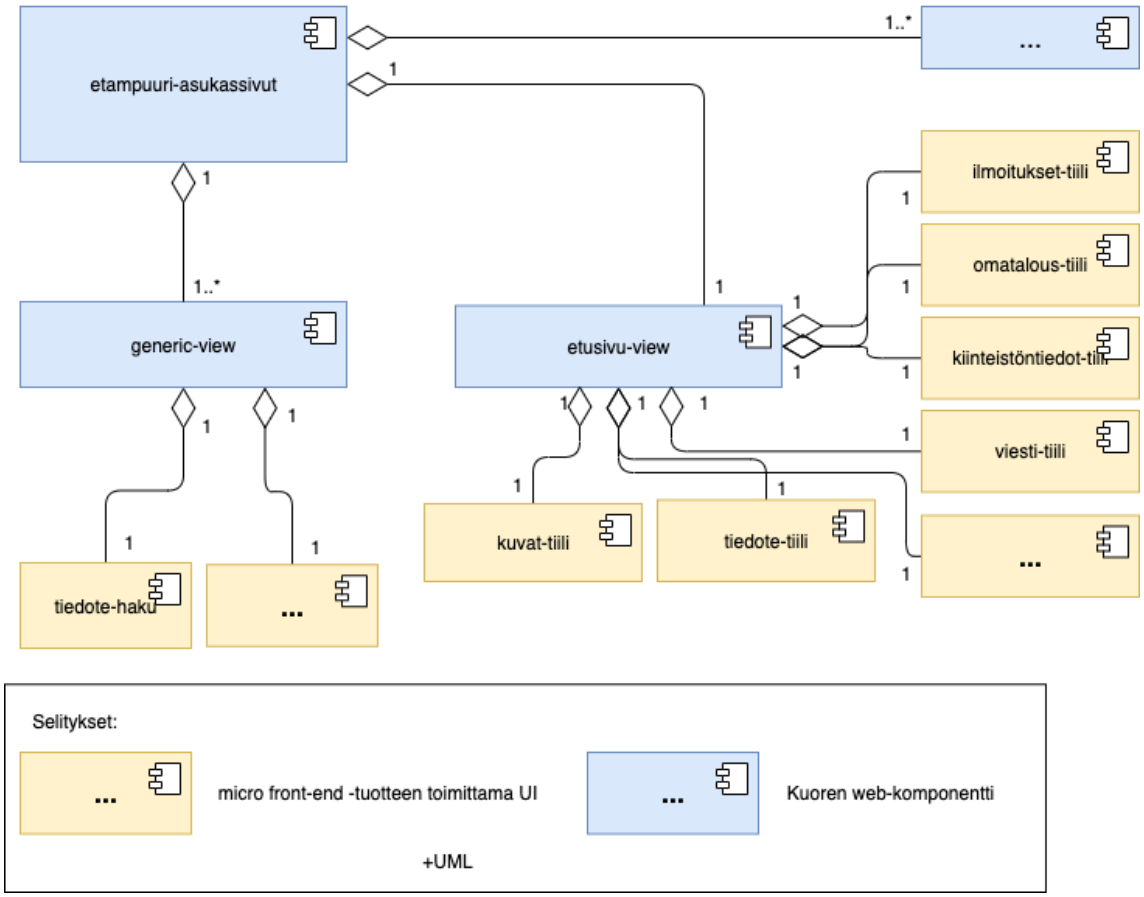


Kuva 3.11. Kuoren ajonaikaista HTML-koodia (Kuva 3.9, Taulukko 3.5, #1). Sovellus koostuu komponenteista ja on myös itse komponentti. HTML:ssä näkyy päätason komponentti *etampuuri-asukassivut*, sen sisällä muita HTML-komponentteja ja muutamia custom element -toteutuksia eli lyhyemmin ”web-komponentteja” (esimerkiksi *vt-sidenav*, *alert-bar ja etusivu-view*).

Päätason komponentin ja tuotteilta ladattavien komponenttien rajapintana toimivat *etusivu-view*- ja *generic-view*-komponentit (Kuva 3.12). *Etusivu-view*-komponentti muodostaa Asukassivujen aloitussivun (Kuvat 2.2 ja 3.14), joka koostuu eri tuotteita edustavista ”tiili”-komponenteista. Se ladataan ensimmäisenä kirjautuneelle käyttäjälle[18]. *Generic-view*-komponentti toimii ”käärönä” (*wrapper*, *adapter*[19]) dynaamisesti ladattavalle tuotteelle. Se muun muassa välittää sivuston kielivalinnan Reduxilta tuotteille

[18] Poislukien tapaukset, joissa kohteita (= kiinteistöjä) on loppukäyttäjän näkymässä useita. Tällöin käyttäjä ohjataan kohdevalinta-sivulle. Muun muassa tätä logiikkaa hallinnoidaan Reduxin koodissa.
[19] https://en.wikipedia.org/wiki/Adapter_pattern

ja vastaavasti kuuntelee tiettyjen konventioiden mukaisia tapahtumia tuotesovelluksista ja välittää ne edelleen *etampuuri-asukassivut*-komponentille.

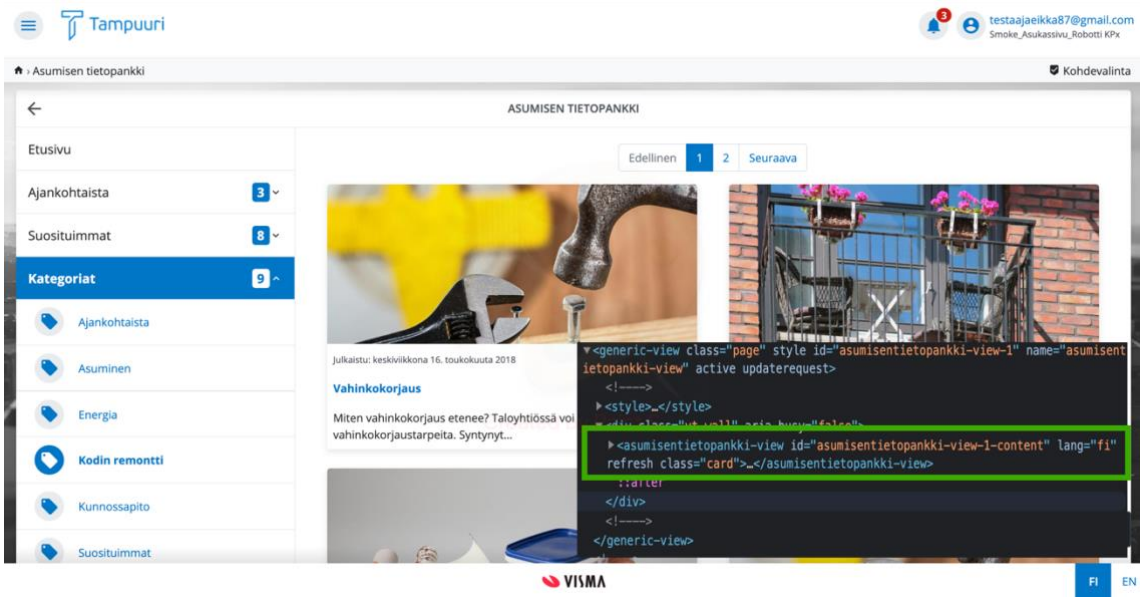


Kuva 3.12. Komponenttinäkymä Asukassivujen rakenteesta. Esimerkkeinä käytetty mm. Ilmoitukset-, Tiedote-, OmaTalous- ja Viesti-tuotteita. Eri tuotteita on käytännössä noin 5-20 kappaletta per asiakas. Kuvassa 3.11 nähdään näistä *etampuuri-asukassivut,* ja sen sisällä *etusivu-view* ajonaikaisessa html-markupissa.

Tuotteet

Jokainen micro frontend -tuote on periaatteessa SPA-sovellus, mutta sillä erolla, että niiden käyttöliittymän koodi suoritetaan toisen sovelluksen, Kuoren, asiakaspäässä. Kuoressa tuotteiden dynaaminen lataus tapahtuu konfiguraation sisältämien ”ohjeiden” mukaisesti: tägin nimi ja osoite tarvitaan, jotta komponentti saadaan käyttöön. Jokainen komponentti rekisteröidään *customElements*-kokoelmaan JavaScriptin globaaliin *window*-olioon sillä hetkellä, kun sen koodia suoritetaan. Latauksen jälkeen komponentille välitetään konfiguraatiosta saadut parametrit (muun muassa rajapintapalveluihin tarvittava avain ja tieto *domainista*, johon kutsu tulee ohjata) ja se liitetään osaksi Kuoren DOMia (Kuva 3.13). Samalla hetkellä käynnistyvät

tuotekomponentin omat elinkaarifunktiot (DOMiin liitettäessä: *connectedCallback*-funktio), jolloin muun muassa näiden omat datahaut käynnistyvät.

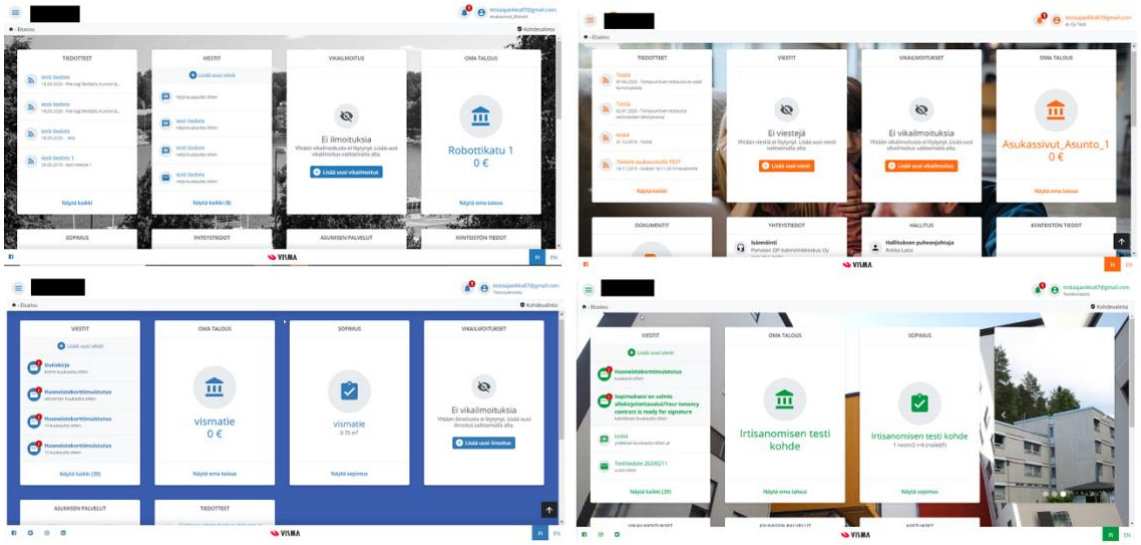


Kuva 3.13. Asumisen tietopankki on yksi Asukassivuille ladattavista micro frontend -tuotteista. Se on myös komponenteista koostua sovellus. Tuote näkyy html-dokumentissa sille määritellyllä tägillä: *asumisentietopankki-view.*

Muokattavuus ja dynaamisuus

Asukassivujen uudistuksessa siirryttiin moniasiakasmalliin tavoitteena silti säilyttää asiakaskohtainen muokattavuus. Ensimmäisessä prototyyppimallissa oli ideana ladata sivuston komponentit konfiguraation mukaan. Tälle mallille perustui sekä itse Kuori, eli sovelluksen päätaso, sekä myös jokainen micro frontend -tuote, jolloin asiakaskohtaiset asetukset voitaisiin määritellä tuotetasolla. Käyttöönottotiimi muokkaa konfiguraatiopalvelun tietosisältöä.

Asiakaskohtaisiin konfigurointeihin kuuluvat paitsi tuotekokoonpano, myös käyttöliittymän teema. Teema muodostuu pääasiallisesti logosta, taustakuvasta sekä kahdesta fontin väristä. Näillä elementeillä käyttöliittymästä saadaan asiakkaan oman brändin mukainen sivusto (Kuva 3.14). Tarvittaessa teemaa voisi laajentaa koskemaan muitakin tyylimäärittelyjä, mutta se toisaalta taas lisäisi käyttöönottotiimin työtä.



Kuva 3.14. Kuvassa neljän eri asiakkaan Asukassivut. Uudet Asukassivut mukautetaan asiakkaiden teemoihin asettamalla logo (peitetty), taustakuva, primäärinen ja sekundäärinen väri (teksti ja kuvakkeet) sekä sosiaalisen median kuvakkeet (vasen alakulma).

Prototyypistä oikeaksi ratkaisuksi

Edellä kuvattiin Asukassivujen kokonaisarkkitehtuuri sekä sen micro frontend -osuus. Toimintamalli pohjautuu esiteltyyn prototyyppitoteutukseen (Kuva 3.3), mutta on luonnollisesti laajempi ja kompleksisempi kokonaisuus: mukana on siis noin 20 micro frontend -palvelua, jotka kommunikoivat yhtä monen mikropalvelun välityksellä Tampuuri APIn kanssa.

Web-komponentit voivat olla sekä hyvin erikoistuneita tai yksikertaisia käyttöliittymäkomponentteja (Kuva 3.2) tai kokonaisia web-sovelluksia (Kuva 3.13), jotka voisivat toimia myös omina SPA-sovelluksinaankin. Asukassivujen micro frontend -arkkitehtuuri toimii siten, että se koostaa noin 20 tällaista sovellusta yhdeksi kokonaisuudeksi. Asiakasyritys ja loppukäyttäjä näkevät vain yhden web-sivuston.

4 Evaluointi

Kehitys ja evaluointi ovat design science -tutkielman keskeiset prosessit. Suunnitteluratkaisuja ja prototyyppejä arvioidaan jatkuvasti kehityksen edetessä. Arvioitavat kohteet iteratiivisessa kehityssyklissä kohdistuvat tiettyyn käsillä olevaan ratkaisumalliin. Valmiin ratkaisun evaluointi on design science -kontekstissa kriittisessä roolissa [HMS04]. Siinä täytyy huomioida sekä toimintaympäristö että ratkaisulle esitetyt tavoitteet ja vaatimukset.

Tässä tutkielmassa esitellään ratkaisuna sekä uusi micro frontend -arkkitehtuurimalli että sen implementaatio: toimiva ohjelmisto. Jälkimmäinen on todiste toimivasta ratkaisusta, mutta jotta ratkaisun kelpoisuutta voidaan verrata muihin esitettyihin malleihin, on syytä evaluoida sitä myös formaalisti.

ATAM (*architecture tradeoff analysis method*) [KKB98, KKC00, CKK01] on ohjelmistoarkkitehtuurien evaluointiin kehitetty metodi, jolla pyritään kartoittamaan arkkitehtuurin riskejä jo ennen varsinaisen kehitystyön alkamista. Metodia voidaan käyttää kuitenkin myös ohjelmistokehityksen myöhemmissä vaiheissa. Sitä on aiemminkin sovellettu kehitysvaiheessa olevien ohjelmistojen arviointiin [JoL01].

Tämän tutkielman aihe on rajattu pääasiassa Asukassivujen micro frontend -osuuteen, mutta kokonaisuus tarvitsee toimiakseen mikro- ja liitännäispalvelut, Tampuuri APIn (Kuva 3.7, Kokonaisarkkitehtuuri) sekä myös toimivan sovellusinfran eli pilvipalvelut. Evaluointiakaan ei siksi ollut mielekästä rajoittaa koskemaan vain tiettyä osa-aluetta.

Tässä luvussa käydään läpi ratkaisulle suoritettu evaluointiprosessi. Aluksi esitellään ATAM-prosessi, sen jälkeen käydään läpi sen käytännön toteutus ja esitellään eri vaiheissa syntyneet tuotokset. Aliluvussa 4.3 näiden tuotosten eli skenaarioiden avulla analysoidaan, kuinka arkkitehtuuriratkaisu onnistuu sille asetetuissa tavoitteissaan. Luvun lopuksi, aliluvussa 4.4, käydään läpi dataa, jota on saatu Asukassivujen tuotantokäytöstä ja käyttöönotoista.

4.1 Architecture tradeoff analysis method (ATAM)

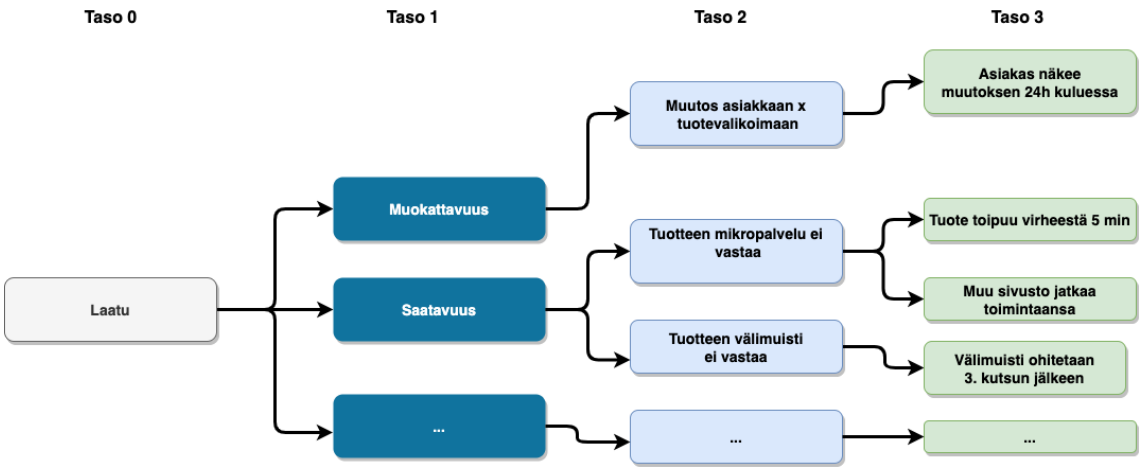
ATAM on muodollinen evaluointiprosessi, jolla pyritään varmistamaan arkkitehtuurin soveltuvuus kehitettävän ohjelmiston perustaksi. Kaksivaiheisen evaluoinnin tarkoitus on löytää liiketoiminnan kannalta keskeiset ajurit ja tärkeimmät laatuattribuutit sekä

tarkastella, kuinka valitut arkkitehtuuriratkaisut tukevat niitä.

ATAM-evaluointiin osallistuvat ohjelmiston sidosryhmät sekä evaluointitiimi. Osanottajien määrä riippuu evaluoitavan ohjelmiston koosta sekä käytettävissä olevista resursseista. Evaluointitiimin koko on 2-5 henkilöä ja sidosryhmät muodostavat 3-11 henkilön joukon[20]. Sidosryhmät koostuvat ohjelmistoprojektin johdosta, arkkitehdeistä, kehittäjistä, testaajista, asiakkaan edustajista ja mahdollisesti muista henkilöistä, joiden näkemyksellä voi olla merkitystä lopputuloksen kannalta.

Evaluointiprosessi on jaettu kahteen vaiheeseen, jotka jakautuvat neljään osioon. Vaiheeseen 1 kuuluvat esittely- ja analyysiosiot, vaihe 2 muodostuu testaus- ja raportointiosioista. Analyysiosioon osallistuvat sidosryhmistä projektinjohto ja arkkitehdit, muihin osallistuvat kaikki sidosryhmät.

ATAM-evaluoinnin tärkeimmät työkalut ovat laatupuu ja skenaariot. Laatupuu muodostaa nimensä mukaisesti puumaisen mallin juurisolmustaan (”laatu”) alkaen (Kuva 4.1). Puun ensimmäiset oksat ovat tunnettuja laatuominaisuuksia (esimerkiksi muokattavuus, saatavuus), toisen tason oksat määrittelevät kutakin ominaisuutta tarkemmin tähdäten kolmannen tason lehtisolmuihin, jotka edustavat mitattavia skenaarioita.



Kuva 4.1. Esimerkki laatupuusta. Juurisolmusta (taso 0) lähtevät oksat edustavat laatuominaisuuksia. Puun lehdet (taso 3) ovat skenaarioita, jotka yhdistyvät tason 2 tarkennuksilla tiettyihin laatuominaisuuksiin.

Mitattavilla skenaarioilla pyritään saamaan arkkitehtuurisuunnitteluun konkretiaa. Ohjelmistoarkkitehtuurilla mahdollistetaan tiettyjen laatuominaisuuksien toteutuminen.

[20] Keskimääräinen kustannus pienen ja keskisuuren ohjelmiston kohdalla [CKK01]

Evaluointivaiheessa voidaan esimerkiksi tunnistaa riskejä, joiden kulkeutuminen myöhempään kehitysvaiheeseen – tai jopa valmiiseen tuotteeseen – voisi tulla myöhemmin paljon kalliimmaksi.

4.2 Asukassivujen ATAM-evaluointi

Asukassivuille tehtiin ATAM-evaluointi 11.-25.3.2021. Evaluointi toteutettiin online-versiona käyttäen presentaatioihin ja keskusteluun Google Chat -sovellusta ja työpajaosuuksiin Mural-ohjelmistoa[21]. Jälkimmäinen toimi digitaalisena työtilana, joka edesauttoi visuaalisuudellaan simuloimaan fyysistä tapaamista, mutta erityisesti se helpotti skenaarioiden äänestämistä.

Tavoitteena evaluoinnille oli saada sidosryhmät osallistumaan ja tuomaan oman näkemyksensä. Tätä tarkoitusta palvelee konkreettisimmin skenaariot, jotka ovat ATAM-evaluoinnin yksi tärkeimmistä työkaluista.

Osallistujat

Evaluointiin osallistui kehitystiimistä kaksi arkkitehtia, tekninen projektipäällikkö (PP), kaksi sovelluskehittäjää, tiiminvetäjä ja testaaja. Käyttöönottotiimistä osallistui kaksi henkilöä. Näiden tiimien ulkopuolisina osallistui projektin omistaja (PO) sekä yrityksen käyttöliittymäsuunnittelija (tässä: kehittäjä).

Toinen arkkitehdeista, tutkielman tekijä, toimi evaluoijana (E). Tiiminvetäjän rooli oli avustaa evaluoinnin fasilitoinnissa, mutta hän omasi myös äänestysoikeuden skenaarioiden priorisoinnissa.

Kehitystiimiin kuuluvat jäsenet ovat luonnollisesti tärkeä sidosryhmä kehitysvaiheessa, mutta tuotanto- ja jatkokehitysvaiheissa myös käyttöönottotiimin rooli kasvaa merkittävästi.

Aikataulu ja osallistuminen

Pvm	Vaihe / Osio	Osallistujat	Sisältö (esittäjä)
11.3.	I / Esittelyosio	Kaikki	Presentaatio ATAMista (E) Esitettiin liiketoimintatavoitteet (PO) Esitettiin arkkitehtuurilliset ratkaisut ja lähestymistavat (E)
17.3.	I / Analyysiosio,	Arkkitehdit, PO, PP	Laadittiin laatupuu ja skenaariot, S1-S8

[21] https://www.mural.co/

	(QAW[22])		Äänestettiin 4 tärkeintä, joita tarkennettiin
25.3.	II / Testausosio	Kaikki	Skenaariotyöpaja[23] Laadittiin skenaariot S9-S22, äänestettiin 10 tärkeintä
	Evaluointi	E	Evaluointi ja tulokset esitellään aliluvussa 4.3
	II Raportointi	E	

Taulukko 4.1. ATAM-evaluoinnin aikataulu, osallistujat ja sisältö.

Vaihe I: Esittely
11.3.2021, online-kokous, 1h

Esittelyosiossa käytiin läpi ATAM-evaluoinnin kulku, termistöä ja skenaarioiden käsitteet. Projektinomistaja esitteli lyhyesti liiketoimintatavoitteet (Taulukko 4.2). Lopuksi esiteltiin yleisellä tasolla arkkitehtuurilliset lähestymistavat ja suunnittelupäätökset.

# Tunniste	Selite
LT1	Infran skaalattavuus helpommaksi. Yksiasiakas vs moniasiakas, suorituskykyä tarpeen mukaan, kohdistetusti
LT2	Käyttöönoton nopeuttaminen kahdesta viikosta kahteen päivään
LT3	Infran kustannusten laskeminen 8%
LT4	Kustannusten nousun kontrollointi: ei lineaarinen suhteessa asiakasmääriin
LT5	Myytävien ominaisuuksien nopeammat toimitukset (koko kehityssykli)

Taulukko 4.2. Liiketoimintatavoitteet ja niiden selitteet.

Käytännössä liiketoimintatavoitteet kohdistuvat skaalautuvuuteen (LT1), kustannuksiin (LT3, LT4) ja nopeampiin toimituksiin (LT2, LT5). Nämä liittyvät vahvasti toisiinsa. Esimerkiksi nopeammat käyttöönotot lisäävät asiakkaiden määrää, mikä aiheuttaa painetta suorituskyvylle ja siten myös infran skaalautuvuudelle.

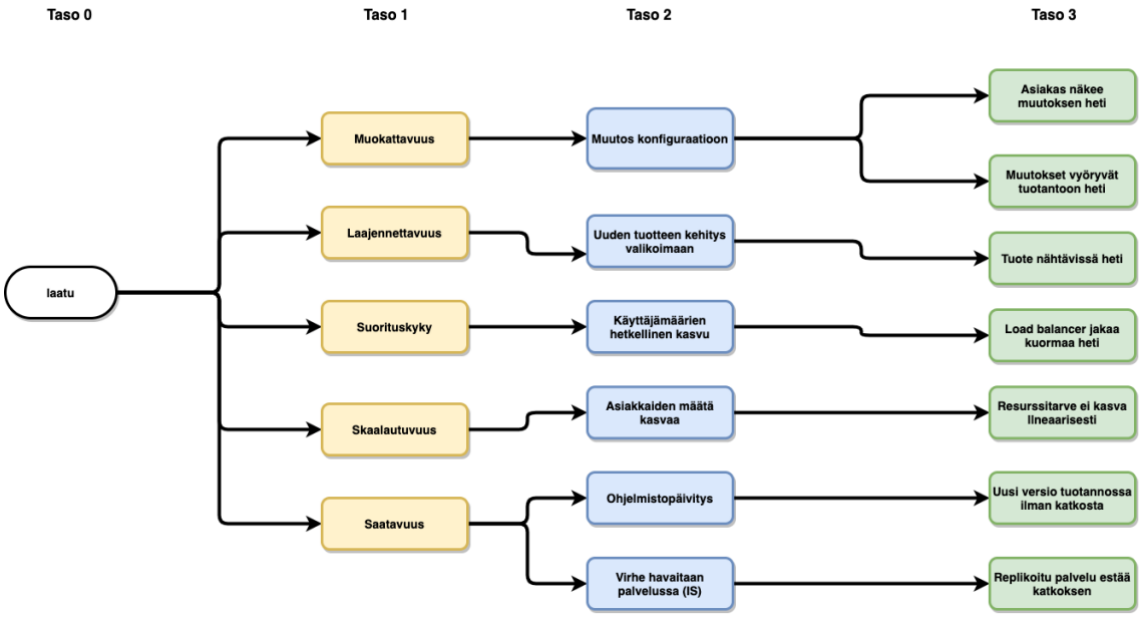
Vaihe I: Analyysi
17.3.2021, online-työpaja (QAW), 2h

Ensin käytiin läpi tavoitteet, jotka olivat laatupuun laatiminen ja skenaarioiden muodostaminen. Skenaarioiden laatimisessa painotettiin sitä, että näkökulma voi olla myös tulevaisuuteen katsova ja tutkiva. Tässä oli ajatus saada osallistujat visioimaan esimerkiksi jatkokehitystä.

Laaditun puun laatuattribuuteiksi nousivat muokattavuus, laajennettavuus, suorituskyky,

[22] ATAM-termistöä: QAW, *quality attribute worshop*
[23] ATAM-termistöä: *scenario brainstorming*

skaalautuvuus ja saatavuus (Kuvat 4.2 ja 4.3).



Kuva 4.2. Analyysiosiossa laadittu laatupuu puhtaaksi piirrettynä. Alkuperäinen, lopullinen laatupuu: Kuva 4.3.

Skenaarioita luotiin 15, joista äänestettiin yksinkertaisella äänestyksellä tärkeimmät. Ääniä oli jokaisella 30% skenaarioiden määrästä eli viisi. Ääniä saaneista kahdeksasta (Taulukko 4.3) poimittiin neljä, jotka jalostettiin tarkemmiksi skenaarioiksi määrittelemälle niiden muun muassa vasteet ja mittarit (Taulukko 4.4, S1-S4).

#Id	**Skenaario**	**Ääniä**
S1	Uusi asiakas tulee Asukassivujen piiriin. Resurssitarve ei kasva lineaarisesti.	**4**
S2	Virhe palvelussa (IS). Replikoitu palvelu estää katkoksen.	**3**
S3	Uusi palvelu tuotevalikoimaan. Palvelu nähtävissä heti	**2**
S4	Kuorma kasvaa. Load balancer jakaa kuorman heti	**2**
S5	Ohjelmiston päivitys. Deployment ilman katkosta	1
S6	Käyttäjällä heikko yhteys. Sovellusta ajetaan välimuistista	1
S7	Asiakkaan teemaa muutetaan. Asiakas näkee muutoksen heti	1
S8	Konfiguraatiomuutos. Uuden konfiguraation vyörytys tuotantoon välittömästi	1

Taulukko 4.3. Vaiheen I analyysiosiossa syntyneet skenaariot.

#	Lähde	Heräte	Ympäristö	Artefakti	Vaste	Mittari	Laatuattribuutit/LT-tavoitteet
S 1	Asiakastilaus	Käyttöönotto perustaa asiakastilin	Normaali toiminta, tuotantoympäristö	Sovellusinfra, konfiguraatio palvelu	Uusi asiakas saa Asukassivut	Resurssitarve **ei kasva** lineaarisesti	Laajennettavuus, muokattavuus / Skaalautuvuus, kustannukset
S 2	Odottamaton virhe (DDoS tms)	Monitorointi havaitsee, ettei palvelu X vastaa	Normaali ympäristö, tuotantoympäristö	Sovellusinfra, palvelu X	Replikoitu palvelu estää katkoksen	**heti**	Saatavuus, tietoturva / Maine, SLA*, skaalautuvuus
S 3	Kehittäjä	Tarve koodata uusi tuote / MF-palvelu	Kehitysympäristö	Koodi, konfiguraatio palvelu	Tuote testattuna ilman katkoja muihin palveluihin	Kolmessa päivässä työn aloittamisesta lähtien	Laajennettavuus, muokattavuus / nopeammat toimitukset (time-to-market)
S 4	Monitorointi, kuormantasaaja	Käyttöpiikki havaitaan monitoroinnissa	Tuotantoympäristö, ruuhkapiikki	Sovellusinfra	Kuormantasaaja jakaa kuorman heti	Vasteaika pysyy normaalina	Saatavuus, suorituskyky, toimivuus, käytettävyys / maine, kustannukset

Taulukko 4.4. Vaiheen I skenaariot täydennettyinä mm. artefaktilla ja laatuattribuuteilla. Tulevaan kehitykseen liittyvät skenaariot merkitty värillisellä taustalla. *) SLA = *service level agreement*, palvelutasosopimus

Vaihe II: Testaus

25.3.2021, online-työpaja (*scenario brainstorming*), 3h

Aluksi esiteltiin ATAM-evaluoinnin tilanne, vaiheen I tuotokset ja tavoitteet tälle työpajalle. Tavoitteena oli kehitellä skenaarioita laajemmalla näkökulmalla ja yhdistää ne lopuksi vaiheessa I laadittuun laatupuuhun. Tähän sessioon osallistuivat kaikki sidosryhmät.

Skenaarioita saatiin 16, joten äänestys tehtiin viidellä äänellä per osallistuja (noin 30% kokonaismäärästä). Skenaarioiden yhdisteleminen osoittautui haastavaksi kokoustilanteessa, joten hyvin samankaltaisia päätyi lopulta äänestykseen. Oleellista oli kuitenkin saada niitä luoduksi. Ääniä saivat lähes kaikki skenaariot (Taulukko 4.5 ja 4.6).

#Id	Skenaario	Ääniä
S9	Palvelun N kapasiteettia nostetaan tuotetasolla reaaliaikaisesti	**7**
S10	Uusi mf-tuote käyttöön asiakkaalle tunnissa.	**6**

S11	AI havaitsee CPU-kuorman kasvun reaaliaikaisesti	**5**
S12	Bugi havaittu tuotannossa. Korjaus tuotannossa ilman palvelukatkosta saman päivän aikana6	**5**
S13	Oletuskonfiguraatio valittavissa asiakastyypin mukaisesti	**4**
S14	Odottamaton virheen lähde voidaan selvittää AI:sta	**3**
S15	Aiemmin toteutettu komponentti käyttöön toisessa (mf-)palvelussa	**3**
S16	Asukassivujen käyttöönotto asiakkaalle. Oletusarvoilla oikeassa URL-osoitteessa.	**3**
S17	Tuote kehitetään parhailla sille valitulla teknologioilla. Ei vaikutusta muihin tuotteisiin.	**3**
S18*	Käyttäjän oikeudet muuttuneet. Rooli-/oikeushallinta ajan tasalla välittömästi	**3**
S19*	Käyttäjän oikeuksiin muutos. Oikeushallinta kontrolloi pääsyä uusien asetusten mukaan välittömästi	**3**
S20	Tuotannossa olevaan tuotteeseen lisätään uusi ominaisuus 3 päivässä	**2**
S21	Yksittäinen tuote testataan CI/CD-prosessissa	**1**
S22	Tuote voidaan toteuttaa alihankkijalla viikossa	**1**
S23	Tuotannossa havaittu haavoittuvuus npm-paketissa. Korjattu kaikkiin tuotteisiin 3 päivässä	**1**

Taulukko 4.5. Ääniä saaneet skenaariot. *) S18 ja S19 ovat lähes samankaltaiset. Tässä varsinaisella sijoituksella ei ole merkitystä, koska molemmat nousivat listalle.

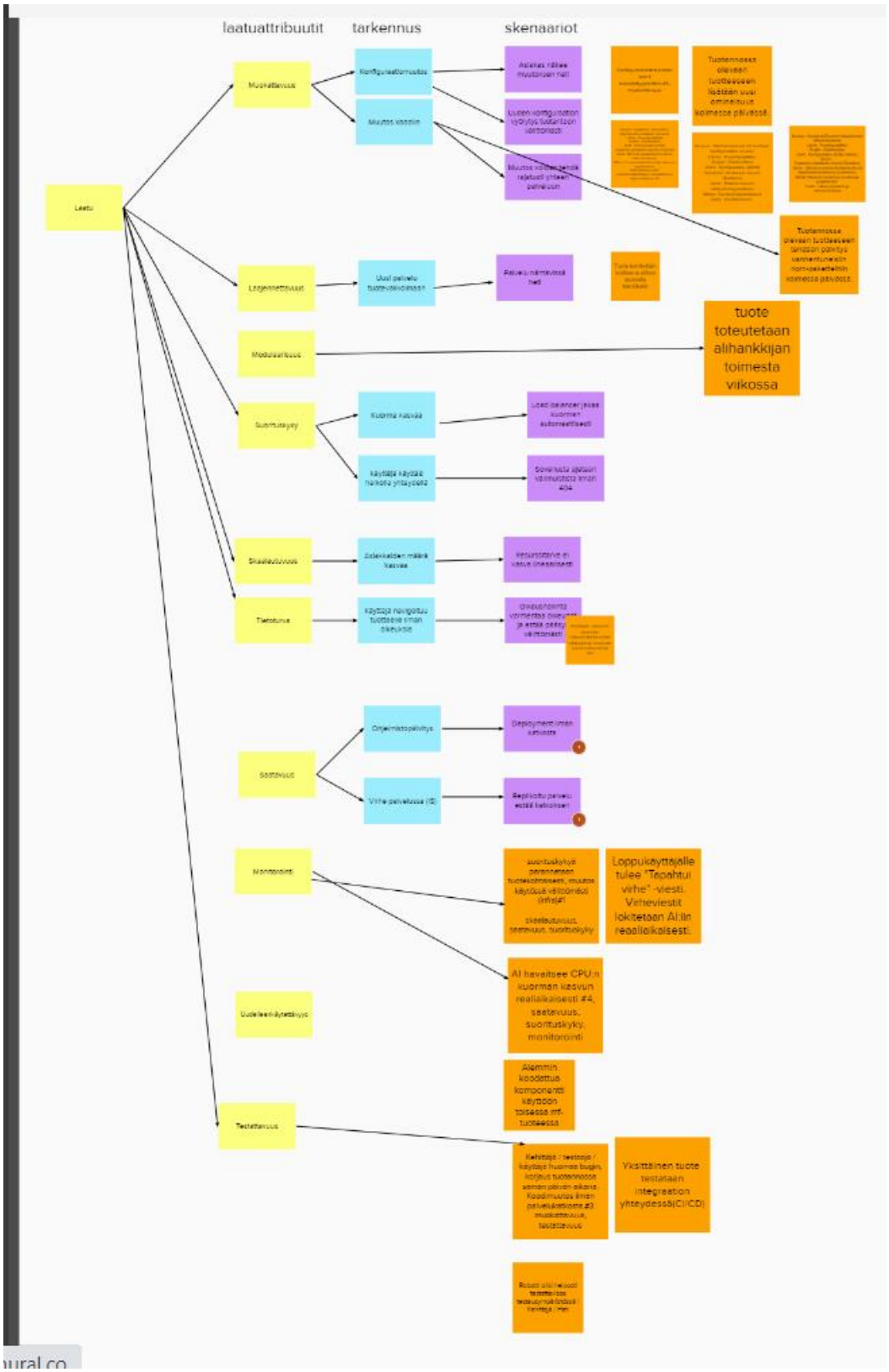
#	Lähde	Heräte	Ympäristö	Artefakti	Vaste	Mittari	Laatuattribuutit/LT-tavoitteet
S9	Monitorointi	Palvelun X kuormituksessa havaitaan nousua	Normaali ympäristö, tuotantoympäristö	Sovellusinfra, palvelu X	Palvelinkapasiteettia nostetaan välittömästi	heti	Skaalautuvuus, saatavuus, tietoturva, monitoroitavuus / Maine, SLA
S10	Käyttöönotto, asiakastilaus	Käyttöönotto konfiguroi tuotteen asiakkaalle	Tuotantoympäristö	Konfiguraatiopalvelu	Mf-palvelu/-tuote näkyy asiakkaalla	1 h	Muokattavuus, / time-to-market
S11	Monitorointi	AI havaitsee CPU-kuorman kasvun reaaliaikaisesti	Tuotantoympäristö	Web-palvelu	Hälytys sähköpostiin tms	reaaliaikaisesti	Saatavuus, suorituskyky, monitoroitavuus, tietoturva / maine
S12	Testaaja, kehittäjä,	Bugi havaittu tuotannossa	Tuotantoympäristö	Koodi	Korjaus tuotannossa ilman palvelukatkosta	saman päivän aikana	Muokattavuus, testattavuus, toimivuus, tietoturva /

	loppukäyttäjä						time-to-market
S13	Käyttöönotto, asiakastilaus	Käyttöönotto konfiguroi asiakkaan asetukset	Tuotantoympäristö	Konfiguraatiopalvelu / konfiguraatio	Oletuskonfiguraatio valittavissa asiakastyypin mukaisesti		Muokattavuus, uudelleenkäytettävyys / time-to-market
S14	Asiakaspalvelu / virheen eskaloituminen	Loppukäyttäjän/asiakkaan havaitsema virhetilanne	Tuotantoympäristö	Sovellusinfra, web-palvelu	Odottamaton virheen lähde voidaan selvittää AI:sta	muutamassa tunnissa	Monitoroitavuus, saatavuus / SLA, maine, kustannus
S15	Kehittäjä	Kehittäjä tekee uutta ominaisuutta A useampaan mf-palveluun	Kehitysympäristö	Koodi	Aiemmin toteutettu komponentti käyttöön toisessa (mf-)palvelussa	Kehitysaikana	Uudelleenkäytettävyys, modulaarisuus / time-to-market
S16	Asiakastilaus, käyttöönotto	Käyttöönotto perustaa uuden asiakkaan	Tuotantoympäristö	Konfiguraatiopalvelu / konfiguraatio	Asiakassivut toimivat *oletusarvoilla* oikeassa URL-osoitteessa.	Välittömästi	Uudelleenkäytettävyys / time-to-market
S17	Tuotehallinta	Uutta tuotetta aletaan suunnitella ja valita sille sopivat teknologiat	Kehitys-, testaus-, tuotantoympäristöt	Koodi	Tuote kehitetään parhailla sille valitulla teknologioilla.	Ei vaikutusta muihin tuotteisiin.	Modulaarisuus, testattavuus, laajennettavuus, muokattavuus /
S18/ S19*	Asiakastilaus	Käyttöönotto muokkaa tuotteelle määriteltyjä rooleja/oikeuksia	Tuotantoympäristö	Konfiguraatiopalvelu	Rooli-/oikeushallinta ajan tasalla	Välittömästi tai muutamissa sekunneissa	Tietoturva, muokattavuus / maine

Taulukko 4.6. Vaiheen II kymmenen eniten ääniä saaneet skenaariot ”jalostettuina”. Tulevaan kehitykseen liittyvät skenaariot merkitty värillisellä taustalla. *) Yhdistetty kaksi samankaltaista skenaariota.

Vaiheessa I laadittua laatupuuta täydennettiin uusilla lehdillä eli ääniä saaneilla skenaarioilla (Kuva 4.3, oranssit muistilaput). Laajempi sidosryhmien osanotto (vaikkakin vaihtelevalla aktiivisuudella) tuotti myös uusia oksia puuhun. Monet skenaariot liittyivät useampaan laatuominaisuuteen, joten on oikeastaan painotuksesta kiinni, mihin ominaisuuteen uusi skenaario katsotaan liitettäväksi. Tämä on vain osoitus laatupuun luonteesta: mitattavat skenaariot edustavat todenmukaisia tilanteita ja niillä voi olla vaikutusta useampaan ominaisuuteen. Aivan kuten herkkyys- ja tasapainokohdilla arkkitehtuurissa.

Uusia laatuominaisuuksia olivat monitoroitavuus, uudelleenkäytettävyys, modulaarisuus, testattavuus ja tietoturva. Modulaarisuus on yksi mikropalveluarkkitehtuurin selkeimmistä ominaispiirteistä. Eniten ääniä saivat skenaariot, jotka liitettiin suorituskykyyn, muokattavuuteen, monitorointiin ja modulaarisuuteen.



Kuva 4.3. Yleiskuva lopullisesta laatupuusta. Tarkoitus tässä on lähinnä visualisoida eri vaiheissa generoituja skenaarioita. Violetit laput ovat analyysiosiossa (Vaihe I) luotuja skenaarioita, oranssit laput testausosion (Vaihe II) tuotoksia. Keltaiset laput edustavat juurisolmua ja laatuominaisuuksia. Kuvakaappaus on Mural-sovelluksen näkymästä.

4.3 ATAM-evaluointi ja suunnittelupäätökset

Laatuominaisuudet

ATAM-evaluoinnin Vaiheessa I laadittu laatupuu ja kehitetyt skenaariot nostivat näistä esille joukon laatuattribuutteja (Taulukko 4.7), joista muokattavuus ja laajennettavuus kohdistuvat suoraan tutkimuskysymyksiin (TK1: nopeammat toimitukset muokattavuuden myötä ja TK2: muokattavuus moniasiakasmallissa).

# tunniste	Selite	Laatuominaisuudet	Skenaariot
LT1	Skaalautuvuus	Skaalautuvuus, suorituskyky	S1, S2*, S9*
LT2, LT5	Nopeammat toimitukset	Muokattavuus, laajennettavuus	S3, S10, S12, S13*, S15, S16
LT3, LT4	Kustannukset	Skaalautuvuus (alaspäin)	S1, S4*, S14

Taulukko 4.7. Liiketoimintatavoitteet, laatuominaisuudet ja niihin liittyvät skenaariot. *) Tulevaan kehitykseen tähtääviä skenaarioita

Vaihe II eli testausosio nosti esille joukon uusia ominaisuuksia (Taulukko 4.8), joista tietoturvaan kohdistui useampia skenaarioita. Muutoin ne jakautuivat varsin tasaisesti.

Laatuominaisuus	Skenaariot
Monitoroitavuus	S11, S14
Uudelleenkäytettävyys	S15, S16
Modulaarisuus	S15, S17
Testattavuus	S12, S17
Tietoturva	S9*, S11, S12, S18/S19

Taulukko 4.8. Vaihe II:n testausosion laatuominaisuudet ja ne esille nostaneet skenaariot.

Skenaariot ja niiden toteutuminen

Seuraavaksi analysoidaan skenaarioiden toteutumista (Taulukko 4.9.). Otetaan tarkasteluun eniten ääniä saaneet skenaariot molemmista evaluointityöpajoista, esitetään näiden toteutumista tukevat suunnittelupäätökset ja arvio skenaarioiden toteutumisen tasosta asteikolla yhdestä kolmeen (1 = ei toteudu, 2 = toteutuu joiltain osin tai on mahdollinen, 3 = toteutuu täysin). Arvo 3 tarkoittaa sitä, että skenaario voisi toteutua täysin sen vasteen mukaisesti. Skenaario S2 on esimerkki, jossa toteutuminen ei ole mahdollista ilman kehitystä pilvialustaan ja/tai sovellusinfraan, siksi sen toteutumaksi on

arvioitu 1. Analyysi-sarakkeessa avataan tiiviisti perustelut ja mahdolliset toteutumiseen vaikuttavat seikat.

#Id	Skenaario	# Suunnittelupäätökset	Toteutuma (asteikolla 1-3)	Analyysi
S1	Uusi asiakas tulee Asukassivujen piiriin. Resurssitarve ei kasva lineaarisesti.	**L1, L2, MF**	**3**	Moniasiakasmalliin siirtyminen poistaa tarpeen kasvattaa ohjelmisto- ja tietokantainstansseja, kuten vanhoilla Asukassivuilla. Konfiguraatiopalvelun käyttö taas mahdollistaa asiakaskohtaisen muokattavuuden. Uusi ratkaisu poistaa kokonaan asiakaskohtaisen tietokantainstanssin tarpeen.
S2	Virhe palvelussa (IS). Replikoitu palvelu estää katkoksen.	**MP/MF**	**1**	Mikropalveluarkkitehtuuri mahdollistaa palveluiden skaalaamisen. Nykyinen toteutus ei vielä salli replikoimista, mutta mahdollistaa esimerkiksi konttiteknologian käytön ja siten tukee myös tätä tulevaisuuteen katsovaa skenaariota.
S3	Uusi palvelu tuotevalikoimaan. Palvelu nähtävissä heti	**L2, MF, SP1, SP2**	**3**	Konfiguraatiopalvelun reaaliaikaisuus sekä MF-arkkitehtuurimalli mahdollistavat tuotekokoelman muokkaamisen ohjelmiston ajon aikana. Dynaamista sisältöä (jota asiakkaan palvelukokoelman muutos edustaa) tukevat SP1-S2. Ratkaisua tukee heräte, jolla käyttäjää informoidaan päivitetystä sisällöstä.
S4	Kuorma kasvaa. Load balancer jakaa kuorman heti	**MP/MF**	**2**	Mikropalveluarkkitehtuuri mahdollistaa pilvialustan toimintojen hyödyntämisen (kuten kuormantasaajan käytön) jo nykyisillä Azuren web app -palveluilla. Jo nyt siis tälle tuki, mutta tällä hetkellä ei vielä konfiguroitu käyttöön.
S9	Palvelun N kapasiteettia nostetaan	**MP/MF**	**1**	Kts. yllä. Jos

	tuotetasolla reaaliaikaisesti			mikropalveluarkkitehtuuria käytetään hyödyntäen esimerkiksi kontteja, voidaan reaaliaikaisesti lisätä resursseja näiden hallinnointiin tarkoitetuilla työkaluilla. Reaaliaikaista resurssien nostoa ei toistaiseksi ole toteutettu, mutta valmius tälle on johtuen mikropalveluarkkitehtuurista.
S10	Uusi mf-tuote käyttöön asiakkaalle tunnissa.	**L2, MP/MF, SP3, SP4**	**3**	Palvelukokoonpanon päivitys tietylle asiakkaalle onnistuu lyhyessä ajassa (L2). MF-tuotteen lataaminen konfiguraation mukaisesti on mahdollista pohjaratkaisun (SP3-S4) ansiosta.
S11	AI havaitsee CPU-kuorman kasvun reaaliaikaisesti	**MP/MF**	**3**	AI (eli Application Insights) on Azuren pilvialustan tuottama monitorointiratkaisu. CPU-kuormaa voidaan tarkkailla mikropalvelutasolla reaaliaikaisesti.[24] Jo nykyinen ratkaisu mahdollistaa erilaisten hälytysten asettamisen, esim. muistin tai CPU:n kulutuksen muutosten johdosta.
S12	Bugi havaittu tuotannossa. Korjaus tuotannossa ilman palvelukatkosta saman päivän aikana	**MP/MF, SP3, SP4**	**3**	Itsenäiset julkaisusyklit mahdollistavat nopeat toimitukset MP-/MF-tasolla. Koko ohjelmistoa ei tarvitse päivittää. Pohjaratkaisut (SP3-S4) tekevät toteutuksesta helpommin hallittavia ja myös testattavia kokonaisuuksia.
S13	Oletuskonfiguraatio valittavissa asiakastyypin mukaisesti	**L1, L2**	**1**	Jokainen MF-palvelu lukee konfiguraationsa konfiguraatiopalvelusta, joten tämä skenaario on täysin tuettu, mutta toteutuminen vaatii käyttöönottotyökalun kehittämistä Tämä on käytännössä toisen tiimin vastuualuetta ja liiketoiminnallinen päätös kohdentaa resursseja sen kehitykselle.
S14	Odottamaton virheen lähde	**MP/MF**	**3**	Samoin kuin S11:n kohdalla: Application Insights (AI)

[24] Application Monitoring for Azure App Service: https://docs.microsoft.com/en-us/azure/azure-monitor/app/azure-web-apps?tabs=net

	voidaan selvittää AI:sta			mahdollistaa virheiden jäljittämisen siihen kytketyistä palveluista. Asukassivut on AI:n piirissä
S15	Aiemmin toteutettu komponentti käyttöön toisessa mf-palvelussa	**SP3, SP4**	**3**	Uudelleenkäytettävyys on pohjateknologian (web-komponentit) perus ideologia. Tämä on koko MF-arkkitehtuurin ydintä tässä toteutuksessa. Asukassivujen kehityksen yhteydessä luotiin myös oma komponenttirekisteri uudelleen käytettäviä komponentteja varten.
S16	Asukassivujen käyttöönotto asiakkaalle. Oletusarvoilla oikeassa URL-osoitteessa.	**L2**	**1**	Tässä skenaariossa tarkoitetaan asiakaskohtaisten *domainien* perustamista Azure-pilvialustalla. Tämä on mahdollista toteuttaa skripteillä, joten sekin voidaan automatisoida.
S17	Tuote kehitetään parhailla sille valitulla teknologioilla. Ei vaikutusta muihin tuotteisiin.	**MP/MF**	**2**	Minkä tahansa teknologian käyttö on mahdollista erillisissä mikropalveluissa. Esimerkiksi liitännäispalveluissa voitaisiin backend toteuttaa Node.js:llä tai MF-palvelu voisi olla React-kirjastoa hyödyntävä - tietyin reunaehdoin (esitelty luvussa 3) Näin ei ole toistaiseksi tehty (siksi 2). Perusteluja käyttää muita kuin valittuja teknologioita ei vielä ole esitetty. Esimerkiksi ulkopuolisen tahon koodaama ohjelmisto voisi olla kandidaatti tälle.
S18/19	Käyttäjän oikeudet muuttuneet. Rooli-/oikeushallinta ajan tasalla välittömästi	**L2, SP1, SP2**	**3**	Micro frontend -tuotepalveluiden sallitut roolit ja oikeudet määritellään konfiguraatiopalvelussa. Konfiguraation muutos astuu heti voimaan, käyttäjä saa herätteen muuttuneesta sisällöstä. Sivun lataus palauttaa ajantasaisen sisällön ja oikeushallinta valvoo jokaista micro frontend -palvelulle tulevaa pyyntöä.

Taulukko 4.9. Skenaariot, suunnittelupäätökset sekä arvio toteutumisesta ja analyysi. Tulevaan kehitykseen liittyvät skenaariot merkitty värillisellä taustalla.

Taulukossa 4.10 esitetään yhteenveto arvosanojen jakautumisesta skenaarioille sekä näiden prosenttiosuus kokonaisuudesta. Skenaarioista täysin toteutuvia on 54%, arkkitehtuuriratkaisun tukemia mutta ei-toteutuvia (toteutuma 2) on 15% ja lisäkehitystä vaativia 31%. Kolmannes siis tärkeimmistä skenaarioista ei toteudu tällä hetkellä.

Toteutuma	**# Skenaariot**	**Osuus kaikista (%)**
3	S1, S3, S10, S11, S12, S14, S15	**54**
2	S4*, S17	**15**
1	S2*, S9*, S13*, S16	**31**
		100

Taulukko 4.10. Toteutumien jakautuminen skenaarioittain ja näiden prosenttiosuudet. *) Tulevaisuutta kartoittavat.

Ei-toteutuvista skenaarioista 75% on tulevaisuutta kartoittavia ja tutkivia skenaarioita, joten näiden painoarvoa voidaan arvioida muun muassa niiden kriittisyyden mukaan. S2 (*Virhe palvelussa (IS). Replikoitu palvelu estää katkoksen*.) on priorisoitu äänestyksessä korkealle, joten sen voidaan ajatella olevan kriittinen tulevan kehityksen kannalta. Käyttäjämäärien kasvaessa palvelujen suorituskykyyn ja saatavuuteen tulee kiinnittää huomiota. Samoin S9 (*Palvelun N kapasiteettia nostetaan tuotetasolla reaaliaikaisesti*) liittyy näihin laatuattribuutteihin.

S13 ja S16 koskevat erityisesti käyttöönottotiimin työkalujen kehitystä, mutta arkkitehtuuriset ratkaisut tukevat näitä molempia. Liiketoiminnallisesta näkökulmasta myös näihin kannattaa panostaa, sillä ne nopeuttavat käyttöönottoprosessia. Tutkimuskysymyksessä (TK1) tarkastellaan käyttöönottojen nopeutumista suhteessa vanhoihin Asukassivuihin, mutta evaluoinnin pohjalta voidaan jo todeta, että uutta ratkaisuakin on mahdollista tehostaa lisää käyttöönottojen osalta.

Arvion 2 saaneista skenaarioista S4 (*Kuorma kasvaa. Load balancer jakaa kuorman heti*) liittyy myös tulevaan kehitykseen ja sovellusinfran kehitykseen. S17 (*Tuote kehitetään parhailla sille valitulla teknologioilla. Ei vaikutusta muihin tuotteisiin.*) tarkoittaa mikropalveluarkkitehtuurin mahdollistamaa palveluiden teknologista riippumattomuutta toisista palveluista. Tämä toteutuu myös micro frontend -palveluiden osalta, kuten luvussa 3 on todettu. Tällaisen tilanteen konkretisoituminen ei toistaiseksi ole näköpiirissä, mutta olisi teoriassa mahdollinen, jos esimerkiksi Asukassivuille haluttaisiin integroida kolmannen osapuolen toteuttama ohjelmistotuote.

Artefaktit, laatuominaisuudet ja suunnittelupäätökset

Seuraavaksi tarkastellaan skenaarioissa esiintyvien artefaktien kautta, kuinka

suunnittelupäätökset vaikuttavat esille nousseisiin laatuominaisuuksiin (Taulukko 4.11). Evaluoinnissa käytetyt artefaktit olivat sovellusinfra, konfiguraatiopalvelu, koodi ja yksittäinen web-palvelu (”palvelu X”, tai ”mf-palvelu”), joka voi tarkoittaa sekä micro frontend -palvelua, mikropalveluita tai jotain liitännäispalveluista.

Artefakti	Laatuominaisuudet	# Suunnittelupäätös (edelliseen vaikuttava)
Sovellusinfra	Skaalautuvuus, saatavuus, tietoturva, monitoroitavuus	L1, L2, MP/MF
Konfiguraatiopalvelu	Laajennettavuus, muokattavuus	L2, SP1, SP2, SP3
Web-palvelu	Saatavuus, suorituskyky, monitoroitavuus, tietoturva	MP/MF (SP3, SP4),
Koodi	Modulaarisuus, testattavuus, laajennettavuus, muokattavuus, uudelleenkäytettävyys	SP1, SP2, SP3, SP4 (SP5)

Taulukko 4.11. Artefaktit, laatuominaisuudet ja niihin liittyvät suunnittelupäätökset.

Sovellusinfra: Jokainen MF-arkkitehtuurin palvelu, mukaan lukien liitännäispalvelut, ajetaan pilvi-infran tarjoamilla resursseilla. Tällä tasolla ei oteta kantaa kooditason seikkoihin, mutta juuri skaalautuvuuden ja saatavuuden näkökulmasta ohjelmiston jako mikropalveluarkkitehtuuriin mahdollistaa resurssien kohdentamisen ja lisäämisen tarpeen mukaan. Hajautetussa palvelussa saadaan myös tarkempaa dataa sovellusinfran tarjoamista monitorointi- ja diagnostiikkapalveluista.

Konfiguraatiopalvelu: Suunnittelupäätökset SP1-SP3 tähtäävät dynaamiseen sivuston sisällön muodostamiseen. Komponenttipohjainen kehitys (SP3) mahdollistaa tehokkaan kooditason ratkaisun käyttöliittymien näkymien hallinnalle esimerkiksi konfiguraatiodatan pohjalta. Tähän tähtäsi jo ensimmäinen prototyyppitoteutus (Luku 3, Kuva 3.3).

Web-palvelu: Mikropalveluarkkitehtuuri ja siten myös sen erikoistapaus, micro frontend -arkkitehtuuri, ovat kehittyneet tarpeesta hajauttaa sovellus pieniin hallittaviin palveluihin. Jokainen palvelu on monitoroitavissa ja kontrolloitavissa itsenäisenä yksikkönään. Näin yksittäiselle palvelulle voidaan esimerkiksi antaa enemmän prosessoritehoa ruuhkapiikin aikaan.

Koodi: Kooditasolla on luonnollisesti vaikutusta kaikkiin ylempiin tasoihin ja tarkemmat suunnittelupäätökset on käyty läpi luvussa 3. SP3 ja SP4 esimerkiksi mahdollistavat MF-

arkkitehtuurin toteutumisen vaikuttaen muun muassa verkkoliikenteen kuormaan: asiakaspään ohjelmistokehys saattaisi vaikuttaa suorituskykyyn nostaessaan verkkoliikenteen määrää. SP3 erityisesti edesauttaa uudelleenkäytettävyyttä.

4.4 Asukassivut tuotannossa

Uudet Asukassivut otettiin ensimmäiselle asiakkaalle pilotointikäyttöön syksyn 2019 aikana. Reilun vuoden jälkeen Asukassivuja on otettu käyttöön noin 60 asiakkaalle. Micro frontend -tuotteita on tässä vaiheessa noin 20, ja liiketoiminnan myötä tuotteiden määrä kasvaa entisestään. Samoin kasvaa tuotekokoonpanojen vaihtelevuus; tuotteiden määrä per asiakas vaihtelee viiden ja reilun 20 välillä (Taulukko 4.12).

(MF = micro frontend)

Asiakasyrityksiä	> 60
MF-tuotteita sivustolla per asiakas (ka)	5 - 23 (10)
Sivustoilla kävijöitä* (01.-31.03.2021)	17 591
Käyttöönotto (tuntia)	7,5 - 15

Taulukko 4.12. Asukassivujen tilanne 04.04.2021. Lähde: Visma Tampuuri Oy.*) Asiakasyritysten asiakkaita.

Asukassivut saadaan toimintakuntoisiksi 7,5 tunnin työllä, jolloin niillä näkyy asiakkaan teema ja niiden perustoiminnallisuudet (esimerkiksi autentikointi) toimivat. Tuotantokelpoiseksi saattaminen riippuu muun muassa asiakkaan tilauksesta: mitä tuotteita on tilattu. Tuotekokoonpano vaikuttaa luonnollisesti käyttöönottoon kuluvaan aikaan. Tähän vaiheeseen kuluu enimmillään lisäksi noin yhden päivän työ. Näiden lisäksi käyttöönottoon liittyy Tampuuri-järjestelmään liittyvää konfigurointia, mutta se rajataan tässä ulkopuolelle, koska siihen ei voida - ainakaan toistaiseksi - vaikuttaa konfiguraatiopalvelusta.

Aiempien Umbraco-pohjaisten sivustojen käyttöönottoihin kuluneesta ajasta ei ole tarkkaa tietoa johtuen tuntikirjausjärjestelmän vaihtumisesta. Näin ollen vertailua vanhan ja uuden ohjelmiston välillä ei voida tehdä suoraan. ATAM-evaluoinnin alussa esitelty liiketoimintatavoite (LT2) antaa kuitenkin suuntaa suuruusluokasta ja tuohon tavoitteeseen päästään. Evaluoinnissa nousi myös esille potentiaalinen kehityskohde (S13), jolla käyttöönoton työtä voisi automatisoida ja siten vielä nopeuttaa tästäkin.

Referenssiarkkitehtuuriksi tuoteperheelle

Noin vuoden käyttökokemuksen jälkeen (11/2020) Asukassivujen kehitystiimille tuli tehtäväksi kehittää saman aikaisesti kahta muuta samaan tuoteperheeseen kuuluvaa

ohjelmistoa. Monimutkaisuudessaan nämä eivät olisi samaa luokkaa kuin Asukassivut. Esimerkiksi tuotteiden lukumäärän osalta vaihtelu olisi pienempää, jolla on tietysti vaikutusta sivustorakenteen ja navigaation luomiseen. Näiden kehitys perustui tässä tutkielmassa esitetylle web-komponenttipohjaiselle micro frontend -arkkitehtuurille.

Uudelleenkäytettävyys monella tasolla oli tässä erittäin tärkeässä asemassa. Ulkoasullisesti yhdenmukaiset käyttöliittymäkomponentit tehtiin omaan komponenttikirjastoon, joten niitä voitiin käyttää kahdessa uudessa sovelluskuoressa ja tuotteissa. Micro frontend -palvelujen kehitys voitiin aloittaa rinnakkain uusien kuorien kehitysten kanssa: näitä voitiin kehittää Asukassivujen kuorikomponenttia vasten.

Käytännössä viisi henkilöä (kaksi arkkitehtia, kaksi kehittäjää ja testaaja) osallistui näiden kahden ohjelmiston yhtäaikaiseen kehitykseen. Molemmat ohjelmistot saatiin testaukseen niille asetetussa tiukassa kahden kuukauden aikataulussa.

Hyvin lyhyessä ajassa tuli myös näyttöä myös sille, että tässä tutkielmassa esitetty arkkitehtuurimalli sopii referenssiarkkitehtuuriksi vastaaville ohjelmistoille.

5 Pohdinta

Tässä tutkielmassa on esitelty ratkaisumalli reaalimaailman ongelmaan design science -kehyksessä. Näin voidaan esittää tietty ratkaisu sitä liiaksi yleistämättä ja keskittyä nimen omaisen toimintaympäristön erikoispiirteisiin ja ongelmakohtiin. Ratkaisun kelpoisuutta arvioitiin ATAM-evaluointimenetelmän avulla sekä myös tuotantokäytöstä saatujen kokemusten ja datan perusteella.

Tässä luvussa esitetään, kuinka esitetty ratkaisumalli ja evaluoinnin tulokset vastaavat tutkimuskysymyksiin. Lisäksi pohditaan teknologiavalinnoista nousseita havaintoja ja potentiaalisia jatkokehityssuuntia. Evaluointia käsitellään omassa aliluvussaan, jossa tuodaan esille siihen liittyviä mahdollisia validiteettiuhkia.

5.1 Vastaukset tutkimuskysymyksiin

TK1: Mahdollistaako uusi arkkitehtuuri nopeammat toimitukset?

Sekä ATAM-evaluointi että kokemukset tuotantokäytöstä osoittavat, että uusi micro frontend -toteutus mahdollistaa nopeammat toimitukset sekä yksittäisten ominaisuuksien kohdalla että koko sivuston käyttöönotossa. Evaluoinnin (Aliluku 4.3, Taulukko 4.9) perusteella skenaariot S3, S10 ja S12, jotka edustavat yksittäisten ominaisuuksien ja/tai korjausten toimitusta, toteutuvat täysin.

Asukassivujen käyttöönotoista saadut kokemukset (Aliluku 4.4) osoittavat, että liiketoiminnalliseen tavoitteeseen (Taulukko 4.2, LT2: *Käyttöönoton nopeuttaminen kahdesta viikosta kahteen päivään*) päästään uudella ratkaisulla. Lisäksi skenaariot S13 ja S16 osoittavat, että tilannetta on mahdollista parantaa entisestään kehittämällä käyttöönoton työkaluja ja automatisoimalla pilvialustan toimintoja.

TK2: Pystytäänkö instanssikohtaisten asennusten räätälöinti korvaamaan moniasiakasmallissa?

Tämä tutkimuskysymys tarkoittaa sekä Asukassivujen ulkoasun muokkausta asiakkaan brändin mukaiseksi että itse sisältöä, eli niitä tuotteita joita asiakkaalle on konfiguroitu käyttöön. Tuotannossa olevat jo yli 60 asiakasta (Aliluku 4.4) omine variaatioineen Asukassivuista osoittavat mallin toimivaksi. Myös ATAM-evaluoinnin skenaarioista muokattavuuteen kohdistui kaikkiaan kuusi skenaariota (Taulukko 4.7), joista neljä edustaa täysin toteutuvia tapauksia.

5.2 Teknologiavalinnat

Ohjelmistokehitys on yleisesti teknologiapainotteista, ja varsinkin web-kehitykselle se on erityisen ominaista: niin HTML-standardi kuin myös JavaScript kehittyvät koko ajan, ja koodia suoritetaan kirjavassa joukossa erilaisia selaimia. Lisäksi erilaisia työkaluja tarvitaan kehityksessä moniin eri vaiheisiin. Luvun 3 suunnittelupäätökset koskivat pitkälti teknologiavalintoja, joissa tuli huomioida käynnissä oleva kehitys ja tulevaisuuden näkymät.

React-kirjasto oli monesti vertailukohtana, ja se olisi jopa voinut olla yksi kandidaatti valituksi pohjateknologiaksi. Jälkeen päin voidaan todeta, että web-komponentit ovat osoittautuneet loistavaksi valinnaksi. Perustelut ovat edelleen samat, mutta nyt näkemys asiasta on vahvempi: hyödynnetään HTML-alustaa vahvemmin sen sijaan, että käytettäisiin kolmannen osapuolen ratkaisuja.

Natiivien web-komponenttien kehitys tapahtuu matalalla tasolla ja on ymmärrettävästi kehittäjäkokemuksensa puolesta varsin heikkoa verrattuina ohjelmistokehysten ja -kirjastojen vastaaviin. Tämän aukon täyttää erinomaisesti LitElement-kirjaston tuoma abstraktiokerros. Kehittäjäkokemuksen lisäksi sen tuoma tehokkuus DOMin renderöinnissä on omaa luokkaansa: siinä missä esimerkiksi React päivittää DOMia muistissa pidetyn virtuaalisen DOMin avulla, päivittää LitElementin käyttämä *lit-html*-kirjasto DOMia suoraan vain sen muuttuneilta osilta.

React-kirjastolle voidaan antaa hyvällä syyllä tunnustusta komponenttipohjaisen kehityksen ja uudelleenkäytettävyyden edistäjänä, mutta etenkin tässä micro frontend -kontekstissa, jossa käyttöliittymään kohdistuu paljon muutoksia suoritusaikana, ja komponentteja ladataan verkon yli, sen käyttö tuntuisi resurssien tuhlaukselta.

Potentiaalisia riskejä ja kehityskohteita

Mainittava on sekin, että uusien teknologioiden käyttöön voi liittyä riskejä. LitElement-kirjasto päätettiin ottaa käyttöön sen ollessa vielä versiossa 0.5.2 ja Asukassivujen kehityksessä oltiin jo suhteellisen pitkällä - ainakin teknologiapäätösten suhteen. Riskinä olisi voinut olla LitElementin kehityksen lopettaminen. ATAM-evaluoinnissa tämä ei enää noussut esille, koska riski (onneksi) jäi toteutumatta.

Uusien teknologioiden käyttö toi myös esille sen, kuinka tärkeä on kehittäjien yhteisö. Parhaiten sellaisen arvon huomaa, kun sitä ei ole. Vaikka web-komponentit toimivat hienosti, niin alkuvaiheessa niiden käsittely web-työkalujen[25] kanssa tuotti melkoisesti ongelmia. Tuntemattomat virheilmoitukset eivät juurikaan tuottaneet hakutuloksia virheitä selvittäessä, eikä ollut tahoja, joiden puoleen kääntyä. Tilanne on muuttunut olemattomasta erinomaiseksi parissa vuodessa. Esimerkiksi Asukassivujen ”rinnakkaisohjelmistot” (Luku 3, aliluku ”Referenssiarkkitehtuuriksi tuoteperheelle”) on rakennettu hyödyntäen aktiivisen Open Web Components -yhteisön[26] suosituksia. Tällaisen yhteisön tietämystä kannattaa hyödyntää jatkossa.

Kuten mikropalveluarkkitehtuuri yleensä, on myös tämäkin ratkaisu kompleksinen. Esimerkiksi muokattavuudella on hintansa: jokainen micro frontend -palvelu hakee omat asiakaskohtaiset konfiguraationsa muistiin konfiguraatiopalvelusta. Samoin jokaisen palvelun täytyy tietää, mistä lähteistä sitä saadaan kutsua[27]. Näitä asioita on toistaiseksi ratkaistu kooditasolla ja sisäisillä kirjastoilla, mutta osa nostettiin esille myös Asukassivujen ATAM-evaluoinnissa.

Yksi skenaarioista liittyi kuormantasaukseen (S4, Taulukko 4.3 ja 4.4). Kehittäjätiimin teknologiaselvityksissä on ollut muun muassa *reverse proxyn*[28] hyödyntäminen, jolloin tälle voitaisiin delegoida kuormantasaus. *Reverse proxy*n avulla voitaisiin myös micro frontend -palvelujen välisestä liikenteestä tehdä luotettua, jolloin myös autentikointiin käytetyn verkkoliikenteen määrää voitaisiin pienentää.

Moni tulevaan kehitykseen liittyvistä skenaarioista liittyi pilvialustan hyödyntämiseen: muun muassa palveluiden replikointiin tai resurssien skaalaamiseen. Mikropalveluarkkitehtuurien yhteydessä puhutaan usein konttiteknologiasta, ja niiden käyttö onkin yleistynyt. Myös Asukassivujen kohdalla niiden käyttö on ollut keskusteluissa mukana ja tulevissa ratkaisumalleissa niiden mahdollisuuksia kannattaa tutkia myös micro frontend -palveluiden kohdalla.

---

[25] Esimerkiksi Webpack ”bundlaukseen” https://webpack.js.org/ ja Babel transpilointiin https://babeljs.io/

[26] Open Web Components https://open-wc.org

[27] CORS-säännöt: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[28] https://en.wikipedia.org/wiki/Reverse_proxy

5.3 Evaluointi

Asukassivuja arvioitiin niiden tuotantokäytöstä saadun datan sekä ATAM-evaluoinnin avulla. Jälkimmäiselle oli kolme tärkeää perustelua. Ensiksi, alkujaan ATAM-arviointi oli tarkoitus suorittaa aikaisemmassa vaiheessa arkkitehtuurikehitystä, jolloin sen tehtävä olisi ollut enemmän riskiteemoja, tasapaino- ja herkkyyskohtia kartoittava katselmointi. Toiseksi, haluttiin tehdä Visma Tampuuri Oy:n ensimmäinen ATAM-pilotointi. Ja kolmanneksi, tarkoitus oli saada validi evaluointi uudelle ratkaisumallille. Aiemmin esitettyjä verrokkiratkaisuja oli arvioitu myös ATAMia käyttäen [HRI17].
Muun muassa aikatauluista (ja jopa maailman tilanteesta[29]) johtuen ohjelmisto ehti tuotantokäyttöön reilusti ennen evaluointia. ATAMia voidaan kuitenkin soveltaa: se voidaan tehdä tiiviissä aikataulussa, se onnistuu myös online-versiona, ja sen painopisteitä voidaan muuttaa. Näin kävi tässäkin tapauksessa: päämääräksi tuli tarkastella Asukassivujen tilaa toisaalta retrospektiivisesti ja toisaalta tulevaa kehitystä silmällä pitäen. Se myös suoritettiin täysin online-versiona.
Haasteita tässä nimenomaisessa evaluoinnissa oli runsaasti. Yhden henkilön suorittamana evaluointi on työläs. Evaluoijan ollessa tuttu henkilö osallistujien motivaatio saattaa olla heikompi, kuin mitä se olisi ulkopuolisen henkilön ollessa vetäjänä. (Pääsääntöisesti osallistuminen oli kuitenkin hyvää ja tuloksia saatiin aikaiseksi.)

Vaikka skenaariot ovat yksinkertaisia, osoittautui niiden laatiminen jossain määrin hankaliksi. Jatkossa niitä voisi yrittää luoda yksinkertaisemmalla mallilla; ehkä pyrkien välttämään ATAM-metodin akateemisempaa sanastoa: lähde, artefakti, heräte ja vaste. Nämä saattavat harvemmin käytettyinä käsitteinä olla harhaanjohtavia ja kenties tehdä yksinkertaisista skenaarioista vaikeammin ymmärrettäviä. Periaatteessa skenaariot ovat tuttuja ohjelmistokehityksen parissa työskenteleville käyttötapausten myötä. Jos skenaarioita lähestytään käyttötapausten kautta, niin ehkä ne tulevat vielä helpommin ymmärrettäviksi. Toisaalta, osallistujien motivaatiolla on tässä kuitenkin merkitystä: jo lyhytkin perehtyminen useaan kertaan esiteltyyn, ja ennalta aiemmin annettuun materiaaliin, tuotti hyvän tuloksen.

Jatkossa evaluointia voisi tarpeen mukaan painottaa laatimalla laatupuusta valmiimman mallin tai arvioimalla arkkitehtuuria valmista testiskenaariokokoelmaa vasten. Laatupuussa voisi olla ainakin mietittynä ensimmäisen tason pakolliset ominaisuudet ja

[29] Covid-19-pandemia 2020-2021

näille valmiit tarkentavat toisen tason ”oksat” tai tyhjät paikat. Vain lehtisolmut tulisi saada luoduksi, täydentämään puun valmiiksi analysointia ja priorisointia varten. Valmista testiskenaariokokoelmaa on hyödynnetty aiemminkin [HRI17], jolloin arkkitehtuurimallien keskinäinen vertailu helpottuu. Tällaisissa tapauksissa evaluoinnin voisi ajatella tapahtuvan yrityksen omien arkkitehtien toimestakin eri vaihtoehtoja punnitessa.

ATAM-evaluointi toteutti oppikirjamaisesti sidosryhmien välisen kommunikaation. Mitä aikaisemmassa vaiheessa tätä tapahtuu, sen parempi. Vaikka tutkielman tekijällä oli suhteellisen selkeä kuva Asukassivujen arkkitehtiuudistuksen alkuvaiheessa tavoitteista, joihin ollaan pyrkimässä, tuli evaluointia valmistellessa esille asioita, joiden kuulemisesta olisi ollut hyötyä jo aiemmin. Esimerkkinä tällaisesta olivat asiakastyyppien eroavaisuudet käytännössä.

Validiteettiuhat

Suurin uhka evaluoinnin validiteetille on tässä tutkielmassa se, että evaluoija on myös tärkeässä roolissa arvioitavan arkkitehtuurin suunnittelussa. Analyysi (Aliluku 4.3) on siten vain yhden henkilön tulkintaa. Tutkimuskysymysten kannalta lienee hyvä asia se, että ratkaisun kelpoisuutta voidaan arvioida myös tuotantokäytön pohjalta (Aliluku 4.4). Näin ollen myös evaluoinnin tulosta voidaan ainakin retrospektiivisten skenaarioiden osalta peilata tuotantokäytöstä saatuihin kokemuksiin.

ATAM-työpajoissa sidosryhmät olivat varsin kohtuullisesti edustettuina, mutta kuitenkin asiakkaan näkökulman olisi voinut olettaa nostavan esimerkiksi käyttötapaustyyppisiä skenaarioita esille. Tämän voisi arvioida pieneksi validiteettiuhaksi. Sen merkitys olisi suurempi, mikäli asiakas olisi selkeästi ohjelmiston tilaajan roolissa ja sillä olisi mahdollisesti kiinnostusta – ja oikeutus – tietää myös arkkitehtuurista.

Asukassivujen kehitystiimiin kuuluu yksi testaaja, mutta voi kysyä, että riittääkö evaluointisessiossa yhden testaajan äänimäärä nostamaan riittävästi esille tämän aihepiirin skenaariot ja sitä myöten laatuominaisuudet. Toisaalta, tämä vastaa suhdetta tiimissä, mutta voisi olla evaluoinnin validiteetille uhka kriittisemmässä ohjelmistossa. Yleisemmin ajateltuna, olisi suotavaa painottaa testattavuutta ja tuoda sitä esille myös muidenkin - etenkin arkkitehtien - toimesta.

6 Johtopäätökset

Mikropalveluita – ja yhä useammin myös micro frontend -arkkitehtuuria – koskevassa kirjallisuudessa näihin liitetään lähes aina usean tiimin [DGL17, EsD16, Fol14, HRI17, Kno16, PMo18, RKU17, PAS20] välinen kehitys. Organisaation kasvaessa monoliittisen ohjelmiston kehitystä jaetaan tiimien kesken ja nämä omistavat kehittämänsä palvelut. Asukassivujen uudistus on kuitenkin esimerkki siitä, että 5 -7 henkilön kehittäjätiimin on mahdollista toteuttaa micro frontend -arkkitehtuuriin perustuva ohjelmisto.

Esitetty ratkaisumalli kyseenalaistaa yleistyksen, jonka mukaan lukumäärältään pieni ryhmä kehittäjiä ei pystyisi hallitsemaan micro frontend -arkkitehtuurin tuomaa kompleksisuutta [PAS20]. Samaa arkkitehtuurimallia on jo ehditty käyttää yksinkertaisemmissa SPA-sovelluksissa Asukassivujen edustamassa tuoteperheessä. Edelleen saman, pienehkön tiimin rakentamana.

Osoitettiin myös se, että moniasiakasmalliin perustuva ohjelmisto voidaan toteuttaa siten, että se säilyttää asiakaskohtaisen muokattavuutensa. Käytännössä vaihtelevien piirteiden hallinta jopa tulee helpommaksi, kun konfiguroitavat asiat määritellään yhdenmukaisella tavalla omaan palveluunsa.

Samoin tämä ratkaisu osoittaa, että vaativaa web-kehitystä on mahdollista ja erittäin kannattavaa tehdä myös ilman kolmannen osapuolen JavaScript-ohjelmistokehyksiä. Web Components -teknologia tullee tekemään jatkossa läpimurron muutenkin uudelleenkäytettävien käyttöliittymäkirjastojen[30] (*design system*) myötä, mutta myös siksi, että ne mahdollistavat uudella tavalla ohjelmistojen sisäiset tuotteet, ”sovellukset sovellusten sisällä”.

Uudistuksen alkuvaiheessa micro frontend -arkkitehtuuri oli suhteellisen tuntematon käsite tieteellisissä julkaisuissa verrattuna mikropalveluihin. Näin on edelleen, mutta toteutusten yleistyessä näidenkin osuus kasvanee. Aihetta käsitteleviin kirjoitukseen Internetissä sen sijaan törmää yhä useammin. Micro frontend -arkkitehtuurin implementaatiot tulevat varmuudella lisääntymään, koska monoliittisia käyttöliittymiä syntyy jatkossakin mikropalveluarkkitehtuuriin siirtymisen tuloksena.

[30] Yksi määritelmä käsitteelle *design system*: https://www.nngroup.com/articles/design-systems-101/

Lähteet

CKK01 P.Clements, R.Kazman and M.Klein, Evaluating Software Architectures: Mehods and Case Studies, Addison-Wesley, 2001.

DGL17 Dragoni N. et al. (2017) Microservices: Yesterday, Today, and Tomorrow. In: Mazzara M., Meyer B. (eds) Present and Ulterior Software Engineering. Springer, Cham. https://doi-org.libproxy.helsinki.fi/10.1007/978-3-319-67425-4_12

ECM15 Standard ECMA-262 6th Edition, June 2015. ECMAScript® 2015 Language Specification. 2015. https://www.ecma-international.org/ecma-262/6.0/ [10.08.2015]

EsD16 D. Escobar et al., Towards the understanding and evolution of monolithic applications as microservices, 2016 XLII Latin American Computing Conference (CLEI), Valparaiso, sivut 1-11. doi: 10.1109/CLEI.2016.7833410, 2016

FoL14 Fowler, M., Lewis, J., Microservices. http://martinfowler.com/articles/microservices.html. [9.7.2019]

Gee19 Geers, Michael. Micro Frontends in Action, 2019. https://livebook.manning.com/book/micro-frontends-in-action/about-this-meap/v-4/ [15.08.2020]

HMS04 Hevner, Alan & R, Alan & March, Salvatore & T, Salvatore & Park, & Park, Jinsoo & Ram, & Sudha,. (2004). Design Science in Information Systems Research. Management Information Systems Quarterly. 28. 75-.

HRI17 Holger Harms, Collin Rogowski, and Luigi Lo Iacono. 2017. Guidelines for adopting frontend architectures and patterns in microservices-based systems. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). ACM, New York, NY, USA, 902-907. DOI: https://doi.org/10.1145/3106237.3117775

IKE18 Experiences Using Micro Frontends at IKEA, 2018. https://www.infoq.com/news/2018/08/experiences-micro-frontends [09.07.2020]

JaC19 Jackson, C. Micro frontends. https://martinfowler.com/articles/micro-frontends.HTML.19.06.2019 [16.07.2020]

JoL01 Jones, Lawrence G.; Lattanze, Anthony: Using the Architecture Tradeoff Analysis Method to Evaluate a Wargame Simulation System: A Case Study. Carnegie Mellon University. Journal contribution. https://doi.org/10.1184/R1/6585809.v1

KKB98 R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson and J. Carriere, "The architecture tradeoff analysis method," *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*, Monterey, CA, USA, 1998, s. 68-78. doi: 10.1109/ICECCS.1998.706657

KKC00 R. Kazman, M. Klein and P. Clements, "ATAM: Method for Architecture Evaluation", *Cmusei*, vol. 4, s. 83, August 2000.

Kno16 Holger Knoche. Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices. In Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE '16). ACM, New York, NY, USA, 121-124. DOI: https://doi-org.libproxy.helsinki.fi/10.1145/2851553.2892039, 2016

LIT20a LitElement, A simple base class for creating fast, lightweight web components, dokumentaatio. https://lit-element.polymer-project.org/ [06.10.2020]

LIT20b Lit-html, An efficient, expressive, extensible HTML templating library for JavaScript, dokumentaatio. https://lit-html.polymer-project.org/ [06.10.2020]

MDN20a MDN web docs, <iframe>: The Inline Frame element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe. 04.07.2020 [13.07.2020]

MDN20b MDN web docs, <video>: The Video Embed element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video 16.05.2020 [24.07.2020]

MDN20c MDN web docs, CustomElementRegistry. https://developer.mozilla.org/en-US/docs/Web/API/CustomElementRegistry. 26.01.2020 [24.07.2020]

MDN20d MDN web docs, Web Components, https://developer.mozilla.org/en-US/docs/Web/Web_Components, 23.07.2020 [19.09.2020]

MDN21a MDN web docs, Web Components, https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM, 12.03.2021 [24.04.2021]

Mic21 Introduction to ASP.NET Core, https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-2.1#build-web-apis-and-web-ui-using-aspnet-core-mvc-1 [30.05.2021]

Nor18 Norton Gray, Polymer Project, Roadmap Update, part 2: FAQ https://www.polymer-project.org/blog/2018-05-02-roadmap-faq. 02.05.2018 [13.09.2020]

OJK19 N. Omer, S. K. Jha and S. Kumar Khatri, "Maintaining Reusable Software Components," *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, Madurai, India, 2019, s. 1350-1352, doi: 10.1109/ICCS45141.2019.9065845.

PAS20 Pavlenko, A., Askarbekuly, N., Megha, S., & Mazzara, M. (2020). Micro-frontends: Application of microservices to web front-ends. *Journal of Internet Services and Information Security*, *10*(2), 49-66.

PMo18 Project Mosaic, Microservices for the Frontend, https://www.mosaic9.org [09.07.2020]

PoP20 Polymer Project, etusivu https://www.polymer-project.org/ [13.09.2020]

PoL20 Polymer Library - versio 3.0, Browser support overview, https://polymer-library.polymer-project.org/3.0/docs/browsers#platform-features, [13.09.2020]

REA20a React, Main Concepts, Components and Props. https://reactjs.org/docs/components-and-props.html [16.07.2020]

REA20b React, Web Components, https://reactjs.org/docs/web-components.html [13.09.2020]

RED20 Redux, A Predictable State Container for JS Apps, API Reference https://redux.js.org/api/api-reference [12.11.2020]

RKU17 D. Richter, M. Konrad, K. Utecht and A. Polze, "Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice," *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Prague, 2017, s. 130-137, doi: 10.1109/QRS-C.2017.28.

SSI20 Wikipedia, Server Side Includes (SSI). https://en.wikipedia.org/wiki/Server_Side_Includes. 16.062020 [13.07.2020]

VIS19 Visma Tampuuri Sähköinen asiointi, Asukassivut. Verkkosivuston markkinointimateriaali https://www.tampuuri.fi/tuotteet/asumisen-verkkopalvelut/asukas-ja-osakassivut/#Asukassivut [02.06.2021]

Zal18 Front-End Microservices, 2018 https://engineering.zalando.com/posts/2018/12/front-end-micro-services.HTML?gh_src=wyv85p?gh_src=wyv85p [09.07.2020]